Rétro-ingénierie des plateformes pour le déploiement des applications temps réel

Rania Mzid





Ecole Nationale des Ingénieurs de Sfax et Université de Bretagne Occidentale

Thèse

présentée pour obtenir le grade de docteur

spécialité : Informatique

par

Rania MZID

Rétro-ingénierie des plateformes pour le déploiement des applications temps réel

---

Thèse soutenue 12 Mai 2014, devant le jury composé de :

Mohamed JMAIEL	**Président**	ENIS
Mohamed KHALGUI	**Rapporteur**	INSAT
Pierre BOULET	**Rapporteur**	Université Lille 1
Jérôme DELATOUR	**Examinateur**	ESEO
Jean-Philippe BABAU	**Directeur**	UBO
Mohamed ABID	**Directeur**	ENIS
Chokri MRAIDHA	**Encadrant**	CEA-LIST

TECHNO-THÈSE CEA-TUNISIE EN CO-TUTELLE ENTRE L'ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'ENIS DE L'UNIVERSITÉ DE SFAX ET L'ÉCOLE DOCTORALE SANTÉ, INFORMATION, COMMUNICATION, MATHÉMATIQUE, MATIÈRE DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte

Les systèmes temps réel embarqués (STRE) sont désormais présents dans plusieurs domaines d'application tels que la télécommunication, l'automobile, l'aéronautique, etc. Devant assurer de plus en plus de fonctionnalités, ces systèmes ont tendance à être de plus en plus complexes [27][46]. En plus de cette complexité, des contraintes concurrentielles liées au marché des systèmes temps-réel embarqués viennent s'ajouter. Ces contraintes obligent les industries à fournir un produit innovant tout en optimisant son coût et son temps de mise sur le marché (*time to market*).

Afin de répondre à cette complexité croissante, l'innovation technologique s'oriente vers une montée en niveaux d'abstraction. Depuis plusieurs années, les couches d'abstraction logicielles ont vu le jour. Notamment, les systèmes d'exploitation ont permis d'abstraire les préoccupations liées à la plateforme matérielle. L'apparition de ces systèmes a permis, d'une part, aux développeurs de se décharger de la complexité du développement qui résulte de l'utilisation des langages machines. D'autre part, leur utilisation facilite le portage des applications sur différents supports matériels. Les systèmes d'exploitation constituent ainsi des supports d'exécution logiciels pour les applications.

Aujourd'hui, les systèmes d'exploitation tendent à leur tour à évoluer et se diversifier. Ceci a paradoxalement complexifié les cycles de développement [50] et a poussé les recherches vers une nouvelle montée en niveau d'abstraction. En effet, les besoins de réutilisation et de portage des applications sur différents systèmes (ou sur plusieurs versions d'un même système), orientent le développement logiciel vers une conception de l'application indépendante de tout système d'exploitation. L'ingénierie dirigée par les modèles (IDM)[76] propose un cadre aux développeurs pour séparer la conception de l'application de son implémentation. Elle prône pour cela l'utilisation des modèles et des transformations des modèles. Les modèles décrivent les systèmes, tandis que les transformations automatisent leur manipulation et leur intégration dans un processus de développement génératif.

Par ailleurs, l'aspect critique des systèmes temps réel embarqués qui résulte principalement des contraintes temps réel exigées par l'application, nécessite de faire appel à des techniques de vérification. L'application de ces techniques permet de prouver qu'un système respecte ses contraintes de temps. Dans ce contexte, l'IDM préconise la vérification des modèles de haut niveau pour une détection plus rapide et efficace des erreurs de développement.

C'est dans ce contexte que cette thèse se situe. Nous nous intéressons au développement des systèmes temps réel embarqués en suivant la ligne de l'IDM. Plus précisément, l'objectif est d'assurer le déploiement d'une application temps réel sur différents systèmes d'exploitation temps réel (RTOS). Ce déploiement doit permettre le portage de l'application d'un RTOS à un autre tout en assurant le respect des contraintes de temps de celle-ci à l'implémentation.

1.2 Problématique

Standardisée par l'OMG, l'approche MDA (Model-Driven Architecture) [65] définit un cadre pour un processus de type IDM, et plus particulièrement pour la mise en œuvre d'une application sur différentes plateformes logicielles d'exécution. En suivant cette discipline, les concepteurs ont la possibilité de décrire leur application dans un modèle de conception indépendant de toute plateforme logicielle (PIM, Platform Independant Model). A partir de ce modèle, plusieurs modèles de l'application spécifiques à différents RTOS doivent pouvoir être automatiquement générés (PSM, Platform Specific Model) à l'aide des transformations de modèles.

En outre, dans une approche MDA, une vérification des contraintes de temps de l'application peut être réalisée au niveau conception (c'est-à-dire au niveau du PIM). A ce niveau, l'utilisation des techniques d'analyse à savoir les techniques d'analyse d'ordonnançabilité [80], a pour objectif d'éliminer (tôt dans le cycle de développement) les défauts de conception. Cela permet de détecter les architectures non faisables menant à des systèmes défaillants c'est-à-dire des systèmes qui ne respectent pas les contraintes de temps de l'application. Toutefois, afin d'être vérifié un modèle doit être exécutable, c'est à dire il doit avoir une sémantique permettant son interprétation. Par conséquent, l'application des techniques d'analyse d'ordonnançabilité nécessite de faire une abstraction de certaines informations de la plateforme logicielle (afin d'obtenir un PIM interprétable). Cette abstraction se traduit en général par des hypothèses sur l'implémentation dans le but de garder la portabilité (l'indépendance par rapport au RTOS) du PIM.

Du fait de la grande variété des RTOS qui existent aujourd'hui, le déploiement de ce PIM sur un de ces RTOS est une tâche qui n'est pas triviale. En effet, même si tous ces RTOS partagent à peu près les mêmes concepts, la caractérisation de ces concepts peut varier d'un RTOS à un autre, d'une famille à une autre ou d'un standard à un autre [5][6]. De ce fait, les hypothèses sur l'implémentation considérées au niveau conception pour vérifier les contraintes de temps, peuvent être non vérifiées pour le RTOS cible. Dans ce cas, le PIM est considéré non-implémentable sur ce RTOS. Par ailleurs, au niveau implémentation, ces hypothèses peuvent avoir une sémantique différente de celle spécifiée au niveau conception. Dans un tel scénario, le PSM résultant n'est pas équivalent au PIM et ainsi les contraintes de temps peuvent être affectées. Pour remédier à ces problèmes, le concepteur effectue une itération sur le modèle de conception (PIM) en le modifiant dans le but de trouver une solution réalisable. Ces modifications sont, d'une part, basées sur l'expérience du concepteur ce qui augmente considérablement le cycle de développement et, d'autre part, réduisent la portabilité du PIM.

1.3 Contributions

L'objectif de cette thèse est de guider le déploiement d'une application temps réel sur différents systèmes d'exploitation temps réel (RTOS) en respectant les principes du MDA,

d'une part, et en assurant la préservation des propriétés temporelles à l'implémentation d'autre part. Pour cela, cette thèse contribue en deux points :

- **L'introduction de la notion de plateforme abstraite pour l'analyse**, plus précisément une plateforme logicielle abstraite pour l'analyse d'ordonnançabilité. Cette plateforme est utilisée au niveau conception pour vérifier les contraintes de temps de l'application et pour préparer la phase de déploiement tout en restant indépendant du RTOS cible.
- **Proposition du processus DRIM** qui permet de guider le raffinement du modèle de l'application indépendant de tout RTOS (PIM) à un modèle de celle-ci spécifique à un RTOS particulier (PSM). Ce processus repose sur une description explicite de la plateforme abstraite d'analyse et du RTOS cible. Il intègre un ensemble de phases intermédiaires entre la conception et l'implémentation permettant ainsi de (1) détecter les modèles de conception (i.e. PIM) non-implémentables pour un RTOS donné, (2) guider l'utilisateur pour le choix d'un RTOS approprié pour une application particulière, (3) effectuer le refactoring du PIM au travers l'application des patrons pour trouver une solution implémentable pour un RTOS donné et (4) générer plusieurs modèles de l'application spécifiques à différents RTOS (i.e. PSM) en assurant le respect des contraintes de temps à l'implémentation. Une méthodologie pour la description des modèles dans DRIM a été présentée et une mise en œuvre de ce dernier pour les systèmes temps réel s'exécutant sur une architecture monoprocesseur a été également exposée.

1.4 Organisation du document

Le manuscrit est organisé selon le plan suivant :

Le deuxième chapitre présente le contexte de cette étude qui aborde le domaine du développement logiciel pour les systèmes temps-réel embarqués. Tout d'abord, nous introduisons les systèmes temps réel embarqués et l'ingénierie dirigée par les modèles (IDM) utilisée pour leur mise en œuvre dans cette étude. Ensuite, nous nous focalisons sur la notion de la plateforme logicielle au sens de l'IDM et nous faisons un tour d'horizon des différents langages proposés dans la littérature pour modéliser cette dernière. Enfin, nous exposons les différentes approches connexes à nos travaux. À partir de cette étude, nous positionnons nos travaux et nous donnons les grandes lignes de nos contributions.

Le troisième chapitre donne un aperçu global des contributions de cette thèse. Nous commençons par expliquer notre contribution pour une prise en compte explicite de la plateforme logicielle, dite abstraite, utilisée pour l'analyse d'ordonnançabilité au niveau conception. Ensuite, nous présentons un processus que nous appelons DRIM (Design Refinement toward Implementation Methodology). Ce processus définit un ensemble d'étapes intermédiaires entre la phase de conception et la phase d'implémentation du flot IDM permettant ainsi le déploiement d'une application temps réel sur différents RTOS.

Dans le quatrième chapitre, nous introduisons un langage pour la description des différents modèles intervenant dans le processus DRIM et nous présentons une méthodologie

pour la description de ces différents modèles en nous basant sur le langage proposé. Les résultats de ce chapitre constituent la base du cinquième chapitre de ce rapport, qui présente une mise en œuvre du processus DRIM. L'objectif de cette mise en œuvre est de détailler les techniques permettant de guider le déploiement des applications visées dans cette étude (i.e. des applications temps-réel s'exécutant sur une architecture mono-processeur et présentant des tâches qui ne peuvent être dépendantes qu'en partageant des données en exclusion mutuelle.)

Le sixième chapitre décrit les expérimentations menées dans cette étude pour évaluer le processus DRIM. Ces expérimentations sont basées sur l'automatisation de ce dernier et une présentation d'un cas d'étude permettant ainsi de vérifier son applicabilité. L'évaluation réalisée a permis non seulement d'explorer les différentes facettes du processus mais elle a montré sa capacité à détecter les problèmes potentiels de déploiement plus tôt dans le cycle d'une part et à suivre la ligne du MDA en garantissant la portabilité de l'application d'autre part.

Enfin, dans le dernier chapitre, nous concluons cette thèse par le bilan des travaux effectués avant d'aborder quelques perspectives à nos travaux.

Chapitre 2

État de l'art

Ce chapitre vise deux objectifs. Il s'intéresse tout d'abord à introduire le contexte de cette étude et les concepts mis à contribution. Il vise ensuite à positionner les contributions de cette thèse par rapport aux travaux existants.

2.1 Introduction

L'objectif de ce chapitre est de positionner les contributions de cette thèse face aux travaux existants. Pour cela, ce chapitre est divisé en trois parties. Dans la première partie, nous débuterons par une introduction aux systèmes temps réel embarqués et l'ingénierie dirigée par les modèles en mettant l'accent sur les concepts mis à contribution dans la suite de cette étude. Cette partie a pour but de familiariser le lecteur avec le contexte d'étude de nos travaux. La deuxième partie de ce chapitre s'intéresse à la notion de plateforme d'exécution et traite en particulier la problématique de modélisation des plateformes. Dans la troisième partie, nous proposons une synthèse permettant de classifier les travaux de recherche qui concernent les processus de développement des systèmes temps réel embarqués. Ainsi, nous distinguons les processus qui se focalisent sur l'aspect temps réel et ceux qui s'intéressent au problème de déploiement. Nous nous intéressons, en particulier, à la considération des plateformes logicielles pour des activités d'analyse ou de déploiement. Au terme de ce chapitre, nous serons alors en mesure de définir les grandes lignes de nos contributions.

2.2 Domaine de l'étude

Cette partie a pour but de familiariser le lecteur avec le contexte d'étude de nos travaux. Pour cela, elle introduit tout d'abord les systèmes temps réel embarqués. Ensuite, elle présente l'ingénierie dirigée par les modèles utilisée pour leur développement. Nous signalons que nous précisons au fur et à mesure les hypothèses que nous avons considérées lors de nos travaux.

2.2.1 Les systèmes temps réel embarqués

Les systèmes temps réels sont des systèmes réactifs. Un système réactif est un système assujetti à l'évolution dynamique d'un procédé qui lui est connecté et qu'il doit piloter en réagissant à tous ses changements d'état dans un temps fini et déterminé [36]. L'architecture classique des systèmes temps réel est illustrée par la figure 2.1. Ces systèmes sont constitués essentiellement de deux sous-systèmes : le procédé à contrôler et le système de contrôle.

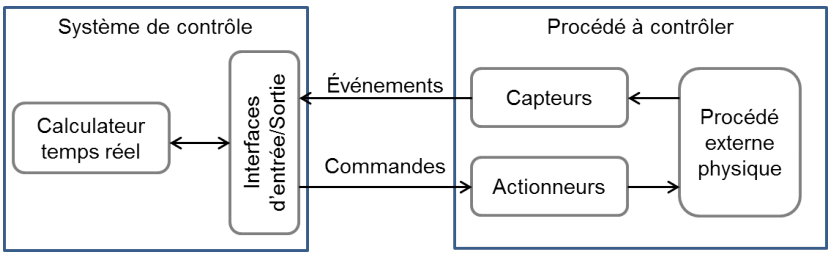


FIGURE 2.1 – *Architecture d'un système temps réel*

Le procédé communique via des capteurs et des actionneurs. Les capteurs récupèrent

des informations du procédé et les transmettent au système de contrôle sous forme d'événements et de données. Les actionneurs commandent le procédé afin de réaliser une tâche bien définie. Le système de contrôle est composé d'un calculateur auquel sont liées des interfaces d'entrées/sorties qui servent à communiquer avec les capteurs et les actionneurs. Le calculateur exécute un algorithme de contrôle en respectant des propriétés temporelles et envoie des ordres de commande aux actionneurs via l'interface de communication. Le procédé contrôlé est souvent appelé l'environnement du système de calcul temps réel. Un système temps réel est dit embarqué lorsque le système de contrôle est enfoui à l'intérieur de l'environnement avec lequel il interagit, comme un calculateur dans un avion ou une voiture. Pour les systèmes temps réel, la validité d'une action dépend du temps qui s'écoule entre la fin de la réalisation de l'action de contrôle et l'événement qui l'a déclenchée. Autrement dit, il est nécessaire que les réactions du système aux événements soient effectuées en un temps inférieur à une borne maximale. Cette borne caractérise une contrainte temps réel. Selon qu'il faille absolument respecter les contraintes ou qu'il soit possible de les violer occasionnellement, ces systèmes sont qualifiés de critiques, fermes ou non critiques. Ainsi, les systèmes critiques nécessitent le respect strict des contraintes. Les systèmes fermes admettent le non-respect de certaines contraintes selon des critères définis par le contexte de l'application. Enfin les systèmes non critiques tolèrent une probabilité de contraintes non satisfaites.

Hypothèse. Dans le cadre de ce travail, nous nous intéresserons aux *systèmes temps réel critiques*. Par conséquent, toutes les contraintes temporelles spécifiées par l'application doivent être systématiquement respectées.

2.2.1.1 Mise en œuvre des systèmes temps réel embarqués

Un système contrôle embarqué se compose : (a) d'une application chargée de piloter le procédé, (b) d'un support d'exécution matériel et/ou logiciel pour permettre l'exécution de l'application.

Architecture matérielle

Les supports d'exécution matériels s'articulent aujourd'hui autour de quatre types d'architectures : (a) monoprocesseur, (b) multiprocesseurs, (c) multi-cœurs et (d) distribuée [85]. Ainsi, lorsqu'une puissance de calcul importante est nécessaire, des architectures multiprocesseurs ou multi-cœurs enrichissent l'approche monoprocesseur d'un parallélisme matériel pour le traitement des données. Par contre, les architectures distribuées sont utilisées pour des installations qui sont écalées géographiquement.

Hypothèse. Dans cette étude, nous supposons que l'architecture matérielle est *monoprocesseur*.

Développement des applications temps réel embarquées

Une application temps réel décrit la réalisation des différentes fonctions de contrôle du procédé physique. Comme nous l'avons déjà mentionné, cette application est soumise, en plus, à des contraintes temporelles. Une technique très répandue pour la prise en compte des contraintes temporelles est la concurrence ou la programmation multi-tâches [22]. En effet, cette technique permet d'exprimer le parallélisme dans une application logicielle. Elle englobe également des techniques pour assurer la communication et la synchronisation entre des entités concurrentes (appelées tâches).

Hypothèse. Dans cette étude, nous admettons que les tâches du système ne peuvent être dépendantes qu'en partageant des données en exclusion mutuelle.

La mise œuvre d'une application multi-tâches sur une architecture monoprocesseur peut suivre une approche logicielle ou mixte. Dans une approche logicielle, tout le traitement est effectué par un programme logiciel s'exécutant sur le processeur. Une approche mixte consiste à mettre en œuvre conjointement (co-design) la configuration matérielle et la programmation logicielle [83]. Cette approche consiste à implanter une partie du traitement sur des circuits spécialisés tels que l'ASIC (Application Specific Integrated Circuit)[82], DSP (Digital Signal Processing)[14] et l'autre partie du traitement sur le processeur lui-même.

Hypothèse. Cette étude s'inscrit dans le cadre de développement des systèmes temps réel embarqués pour une composante matérielle donnée. Toutes les fonctionnalités de l'application sont supposées être *mises en œuvre de manière logicielle*.

Afin de faciliter la mise en œuvre d'une application multi-tâches sur une architecture monoprocesseur, un support d'exécution logiciel (système d'exploitation temps réel)[87] offre des services d'exécution tout en masquant les spécificités du matériel sous-jacent.

Hypothèse. Dans cette étude, nous nous intéressons à mettre en œuvre des applications logicielles dont l'exécution repose sur un *système d'exploitation temps réel*. Les systèmes d'exploitation envisagés sont supposés disponibles. Cependant aucun système particulier n'est imposé.

Système d'exploitation temps réel

Un exécutif temps réel ou système d'exploitation temps réel (appelé en anglais Real-Time Operating System ou RTOS) est composé d'un noyau temps réel et de modules ou bibliothèques complétant le noyau et facilitant le développement de l'application. Un système d'exploitation est dit temps réel lorsqu'il implémente les mécanismes et les services qui, si utilisées correctement peuvent garantir les délais souhaités. Pour la prise en compte des aspects temps-réel, un exécutif temps réel doit en particulier :

- gérer la concurrence par l'adoption d'une stratégie pour partager le temps processeur, à travers des techniques d'ordonnancement temps-réel, entre les différentes tâches en attente d'exécution ;
- gérer l'accès aux données et aux ressources partagées à travers des techniques de synchronisation adaptées aux contraintes au temps-réel ;

- offrir des mécanismes de communication adaptés au temps-réel entre les entités concurrentes du système de contrôle ;
- gérer les horloges pour la manipulation du temps ;
- assurer une gestion prédictible de la mémoire

Hypothèse. Nous nous intéressons aux différents artéfacts du noyau temps réel liés à *la concurrence*, a*u partage de ressources logicielles* et *à la gestion du temps*, que nous supposons suffisants pour la réalisation des applications visées par cette étude.

Il existe aujourd'hui dans la littérature trois principaux standards pour les RTOS : POSIX [5], OSEK/VDK [6] et ITRON[3]. En général, un noyau temps réel peut être conforme à un de ces standards, il peut être commercial tel que VxWorks [2] ou open source comme RTEMS [8]. Dans tous les cas, un noyau temps réel doit accomplir les fonctionnalités décrites ci-dessus. Cependant, les mécanismes mis en œuvre pour la réalisation de ces fonctionnalités peuvent varier d'un noyau à un autre. L'ordonnanceur est l'élément de base d'un noyau temps réel, qui permet de planifier l'exécution des différentes tâches (déterminer l'ordre d'allocation des tâches au processeur). Un ordonnanceur est caractérisé par sa politique d'ordonnancement citefixed [93] [54] et son mode (préemptif ou non préemptif). Selon le noyau, une ou plusieurs politiques sont supportées et un des deux modes (ou les deux) est possible. Ces différents paramètres qui caractérisent l'ordonnanceur ont un impact direct sur l'exécution de l'application et par la suite sur les propriétés temporelles.

2.2.1.2 Vérification des applications temps réel embarquées

Les applications temps réel embarquées possèdent deux aspects : un aspect fonctionnel et un aspect non-fonctionnel. Par conséquent, la vérification de ces applications passe par la vérification de ces deux aspects. Pour l'aspect fonctionnel, la vérification consiste à vérifier que le système produit un résultat correct pour chaque ensemble de données en entrées [35][12] [39]. En ce qui concerne l'aspect non-fonctionnel, la vérification consiste à assurer que le système respecte les contraintes non-fonctionnelles telles que les contraintes temporelles, les contraintes de consommation mémoire ou d'énergie. Parmi les techniques de vérification de l'aspect non-fonctionnel, on distingue les techniques statiques basées sur l'analyse et les techniques dynamiques basées sur la simulation ou le test.

Hypothèse. Dans cette étude, nous nous intéressons à la vérification de *l'aspect non-fonctionnel* en particulier les *contraintes temporelles* en utilisant les *techniques d'analyse statiques*. Nous nous visons plus précisément *l'analyse d'ordonnançabilité*.

Techniques d'analyse d'ordonnançabilité

Les techniques d'analyse d'ordonnançabilité permettent de vérifier si une configuration donnée de tâches est ordonnançable, c'est-à-dire que leur charge de calcul *(load)* est réalisable sur la plateforme cible en respectant les contraintes temporelles qui leur sont affectées. En effet, l'exécution de chaque tâche doit respecter un délai maximum définissant ainsi une

échéance à ne pas dépasser. L'échéance d'une tâche est l'instant auquel l'exécution de la tâche doit se terminer. Le dépassement d'une échéance constitue une faute temporelle. La vérification des échéances est fortement liée à l'algorithme d'ordonnancement temps réel utilisé pour créer cette configuration de tâches. En fait, les algorithmes d'ordonnancement permettent d'établir « au mieux » les niveaux de priorité des tâches pour un contexte applicatif donné.

L'analyse d'ordonnançabilité offre deux techniques pour vérifier la faisabilité d'un système : une technique basée sur les tests de faisabilité et une technique basée sur le calcul de bornes maximales sur les temps de réponse des tâches [23]. La première technique consiste à définir des conditions nécessaires et /ou suffisantes applicables sous certaines hypothèses du système. Ces conditions font en général appel à une description simple de l'application. Par exemple, selon [54], un système ordonnancé en utilisant Rate Monotonic (RM) ou Deadline Monotonic (DM) est faisable si et seulement si :

$$\sum_{i=1}^{n} \frac{c_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \tag{2.1}$$

Dans cette expression C_i définit le temps d'exécution d'une tâche T_i ; P_i est sa période d'activation et n étant le nombre de tâches. On note par D_i l'échéance de la tâche T_i. Cette condition est applicable si toutes les tâches applicatives sont périodiques et indépendantes et que l'échéance de chaque tâche est égales à sa période $(P_i = D_i)$. Pour des descriptions plus élaborées de l'application, la deuxième technique consiste à calculer une borne maximale du temps de réponse de chaque tâche et à vérifier si cette borne est inférieure à son échéance. Si les temps de réponse de toutes les tâches applicatives sont inférieurs à leurs échéances, on peut confirmer que l'application respecte ses contraintes de temps. Dans ce cas, la formule de calcul du temps de réponse dépend de l'application et de la politique d'ordonnancement considérée. Par exemple si toutes les tâches sont indépendantes et pour un ordonnanceur préemptif à priorité fixe [52], le temps de réponse est la plus petite solution de l'équation :

$$Rep_i = c_i + \sum_{T_j \in HP} \left\lceil \frac{Rep_i}{P_j} \right\rceil * c_j \tag{2.2}$$

R_i représente le temps de réponse de la tâche T_i et HP_i correspond à l'ensemble des tâches ayant des priorités supérieures ou égales à T_i. Dans le cas où les tâches sont dépendantes par partage des ressources logicielles (gestion d'accès aux sections critiques), un nouveau terme B_i s'ajoute à la formule précédente.

$$Rep_i = c_i + B_i + \sum_{T_j \in HP} \left\lceil \frac{Rep_i}{P_j} \right\rceil * c_j \tag{2.3}$$

B_i représente le temps de blocage maximal, il correspond au temps pendant lequel une tâche reste bloquée en attendant la libération d'une ressource exclusive par une ou plusieurs tâches

de priorité inférieure.

Analyse d'ordonnançabilité et protocoles de synchronisation

Le partage de ressources dans un système temps réel peut être à l'origine de deux problèmes : inversion de priorité et inter-blocage. Une inversion de priorité survient lorsqu'une tâche est bloquée par une tâche de priorité moindre ne partageant pas de ressource avec la tâche plus prioritaire. On ne peut alors plus prévoir le temps de blocage de la tâche de plus haute priorité. Figure 2.2 ci-dessous illustre une situation d'inversion de priorité pour un système de trois tâches tel que $p_1 > p_2 > p_3$ et une ressource R_0 partagée entre T_1 et T_3. Dans ce cas l'exécution de la tâche T_1 qui est la plus prioritaire est bloquée par la tâche T_2 qui est moins prioritaire. Le temps de blocage supplémentaire du à l'inversion de priorité est $t_5 - t_4$.

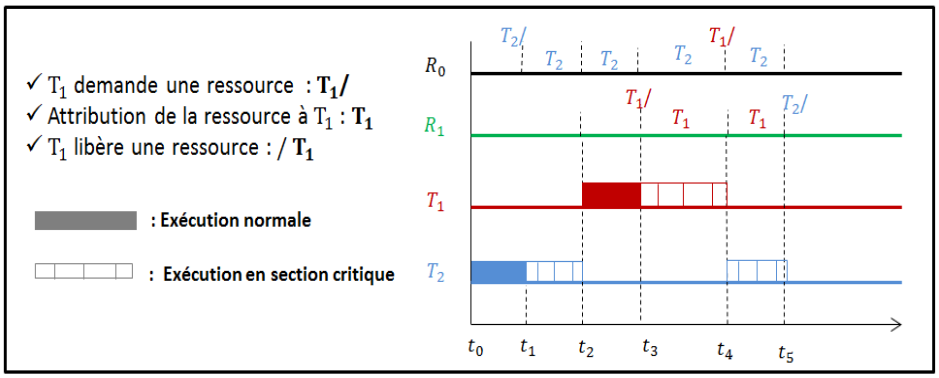


FIGURE 2.2 – *Situation d'inversion de priorité entre T_1 et T_2*

Un inter-blocage survient si les tâches du système partagent plus d'une ressource à la fois. La Figure 2.3 illustre une situation d'inter-blocage pour un système de deux tâches tel que $p_1 > p_2$ qui partagent deux ressources R_0 et R_1. Dans ce cas, les deux tâches sont en inter-blocage car la tâche T_1 qui est la plus prioritaire demande la ressource R_0 pour terminer son exécution par contre cette ressource est utilisée par T_2. De même T_2 ne peut pas prendre R_1 pour terminer son exécution. Pour résoudre ces problèmes (inversion de priorité et inter-blocage), une solution classique est d'utiliser les protocoles où les conflits sont résolus en réalisant des ajustements sur les priorités des tâches lors de l'accès à la section critique. Les protocoles les plus connus sont Priority Ceiling Protocol (PCP) [43] et Priority Inheritance Protocol (PIP) [81] : Dans le protocole PIP, la tâche qui utilise la ressource hérite de la priorité de la tâche qui demande la ressource et qui se trouve en tête de liste dans la file d'attente. Ceci permet de diminuer le temps de blocage de la tâche la plus prioritaire en inhibant la préemption de la tâche qui possède la ressource par des tâches de priorité moyenne. Ce protocole est capable de gérer juste l'inversion de priorité sans résoudre le problème d'inter-blocage. Le protocole PCP suppose que chaque tâche possède une priorité fixe et que les ressources utilisées sont connues avant le début de l'exécution. Chaque ressource est caractérisée par

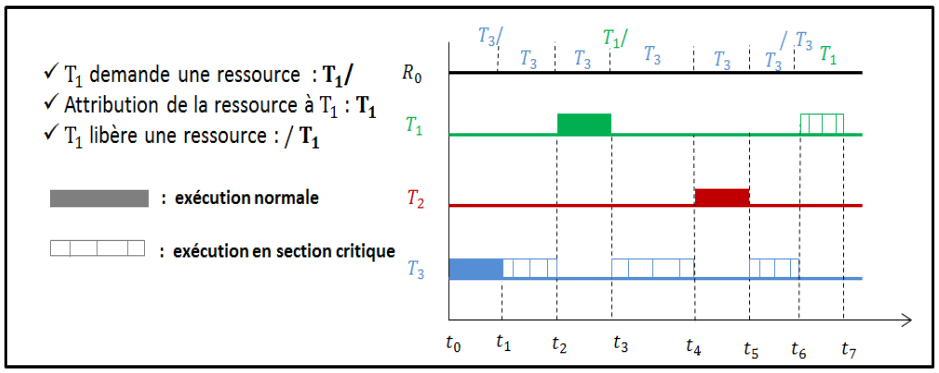


FIGURE 2.3 – *Situation d'inter-blocage*

un plafond de priorité (priority ceiling) qu'on note π_i. On définit aussi le plafond de priorité courant du système que l'on note $\gamma = Max_i(\pi_i)$. Si aucune ressource n'est utilisée $\gamma = \delta$ qui est une priorité plus basse que n'importe quelle priorité dans le modèle. Une T_i tâche ne peut prendre une ressource libre R_j que si la priorité de la tâche T_i est strictement supérieure à γ. Dans le cas échéant, la tâche T_i est bloquée et la tâche T_j qui bloque cette dernière hérite la priorité de T_i. La tâche T_j est exécutée avec cette priorité jusqu'à ce qu'elle libère toutes les ressources ayant un plafond de priorité supérieur à la priorité de tâche T_i. L'utilisation de ce protocole permet d'éviter les deux problèmes mentionnés (inversion de priorité et inter-blocage). Le calcul du temps de blocage dépend fortement du protocole utilisé pour accéder à une ressource partagée. La formule de calcul du temps de blocage dans le cas où le protocole PCP est utilisé est donnée ci-dessous :

$$B_i = \max_{T_j \in HP, R_k \in R} \{U_{R_k,T_j} : p_j < p_i \; and \; \pi_k \geq p_i\} \tag{2.4}$$

Dans cette expression, R est l'ensemble des ressources qui constituent le système et $C_{(R_k,T_j)}$ correspond à l'utilisation de la ressource R_k par la tâche T_j. Le temps de blocage B_i représente la durée de la plus grande section critique parmi celles des tâches de priorités inférieures ou égales à p_i et dont la ressource relative R_k est telle que π_k est supérieure ou égale à p_i.

De part de leur nature, les systèmes temps réel embarqués requièrent un processus de développement rigoureux afin d'atteindre la correction fonctionnelle et temporelle. La partie qui suit introduit alors une ingénierie émergente qui offre des solutions pour le développement de tels systèmes, c'est l'ingénierie dirigée par les modèles.

2.2.2 Ingénierie Dirigée par les modèles (IDM)

L'ingénierie dirigée par les modèles (IDM) propose l'amélioration du développement de systèmes complexes en permettant de se concentrer sur des préoccupations plus abstraites que la programmation classique au travers de modèles. Cette ingénierie offre un cadre méthodologique outillé aux développeurs des systèmes temps réel embarqués, qui se concentre

désormais sur l'élaboration des modèles abstraits, plutôt que sur des concepts liés à l'algorithmique et la programmation. Cette partie introduit cette nouvelle technologie en se focalisant sur les aspects importants pour cette étude.

2.2.2.1 Modèles, méta-modèles et transformations de modèles

De nombreux travaux [24] [16] [77] [37] se sont intéressés à l'IDM, ce qui a permis de clarifier les concepts liés à celle-ci. En nous basant sur ces travaux, nous définissons succinctement dans ce qui suit les notions de base de l'IDM.

Modèles

Un modèle est une représentation abstraite d'un système réalisée dans une intention particulière.Un modèle peut représenter un système dans sa totalité (la structure, le comportement et les propriétés non-fonctionnelles) ou bien il peut représenter juste un aspect du système en occultant ces autres aspects. Les différentes étapes d'un flot de développement d'une application nécessitent différents types de modèles définis avec une précision adaptée et contenant des informations pertinentes pour l'utilisation qui en est faite.Dans un cadre IDM, un modèle doit être interprétable(non-ambigu)afin qu'il puisse être manipulé de manière automatique(outillage).Ceci n'est possible que si ce dernier est exprimé dans un langage clairement structuré et interprétable, appelé méta-modèle.

Méta-modèles

Un *méta-modèle* est un modèle qui définit un langage de modélisation [69].Un méta-modèle définit précisément les concepts d'un langage de modélisation ainsi que les relations entre ces concepts.Un méta-modèle est écrit dans un langage appelé méta-langage. Un modèle bien formé est *conforme* à son méta-modèle.A partir d'un *modèle bien formé*, il est possible de le transformer en un autre modèle ou bien de générer du code ou de la documentation. Cependant, la vérification de cette relation de conformité est importante avant toute transformation ou génération. En effet, il est nécessaire de s'assurer qu'un modèle est syntaxiquement et sémantiquement conforme à son méta-modèle avant de produire une application à partir de celui-ci.

Transformations de modèles

La transformation de modèles est un processus de conversion d'un ensemble de modèles d'une application donnée à d'autres modèles de la même application [65]. Une transformation de modèles définit un ensemble de règles pour passer d'un modèle source conforme à un méta-modèle source à un modèle cible conforme à un méta-modèle cible. Ces règles sont définies au niveau du méta-modèle et seront exécutées sur les modèles pour passer d'un modèle à un autre. La transformation de modèles peut être du type transformation de modèle

en modèle dans le but de raffiner ou changer (refactoring [59]) le modèle source. Ce type de transformation peut être modélisé par les langages de transformations de graphes tels que QVT (Query/View/Transformation) [67] ou (Atlas Transformation Language) ATL [47]. La transformation de modèles peut aussi être de type transformation de modèle en texte dans le but de générer du code ou de la documentation par exemple.

Hypothèse. Seules les *transformations de modèle à modèle* sont utilisées dans cette étude.

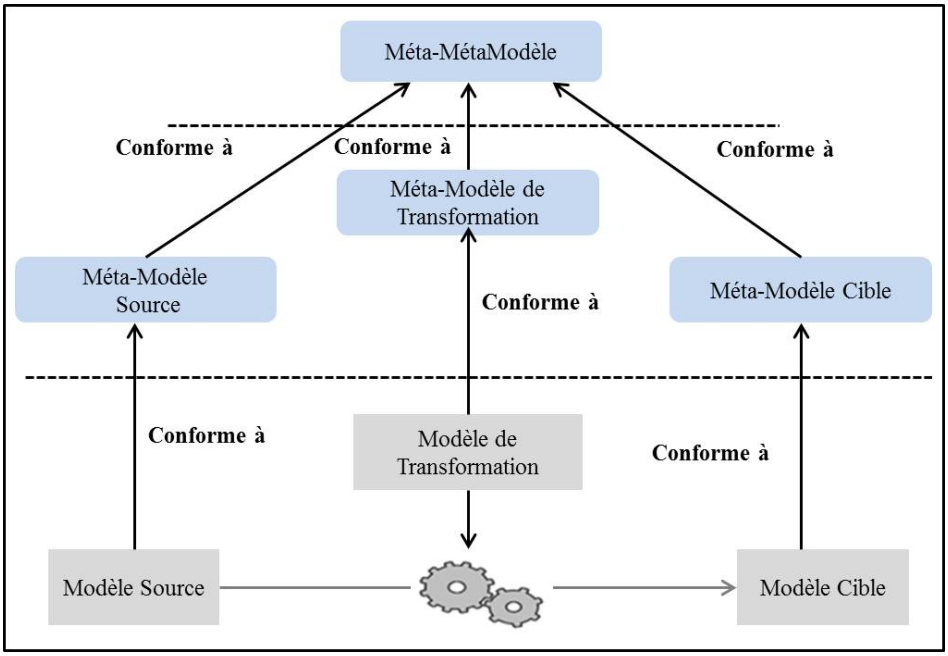


FIGURE 2.4 – *Transformation de modèles*

2.2.2.2 Approche MDA

A partir de ces principes très généraux, plusieurs approches ont été proposées [65] [26] [84] pour intégrer les notions de l'IDM dans un cadre méthodologique et mettre en œuvre ses principes.Parmi ces approches, nous nous intéressons dans cette étude à l'approche MDA (Model Driven Architecture) [65] sur laquelle repose ce travail. Lancé en 2000 par l'OMG, le MDA vise à améliorer la portabilité, la maintenance et la réutilisation des modèles en préconisant l'utilisation de plusieurs types de modèles. En faisant une séparation claire entre les modèles de l'application indépendants des plateformes et les plateformes elles-mêmes, le MDA permet le déploiement d'une même application sur différentes plateformes. Dans un cadre MDA, ce déploiement se concrétise par une transformation de modèles ayant comme entrées le modèle de l'application indépendant de toute plateforme technologique (PIM) et le modèle de plateforme cible (PDM) et comme sortie le modèle de l'application spécifique à la plateforme correspondante (PSM). Le principe clé de l'approche MDA pour la mise en œuvre des principes de l'IDM consiste à s'appuyer sur l'utilisation des standards. Pour cela,

le MDA préconise l'utilisation du standard UML pour la description des différents modèles et méta-modèles et sur le standard MOF (Meta Object Facility) [69] pour la description des méta-métamodèles.

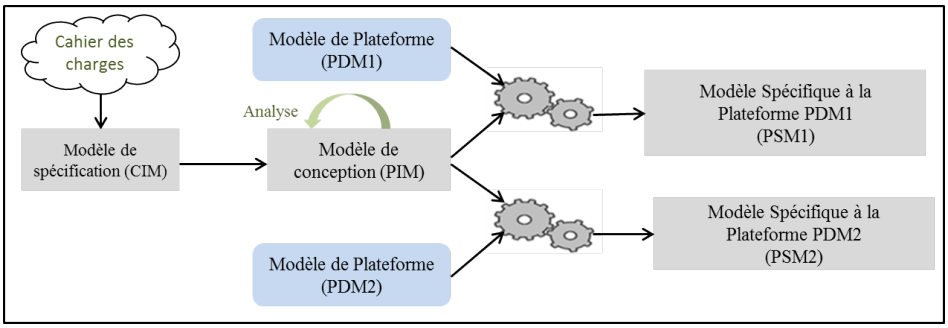


FIGURE 2.5 – *Processus de développement des systèmes temps réel embarqués dans un cadre MDA*

Pour les systèmes temps réel embarqués, le processus de transformation des exigences en un système final en suivant une approche MDA est constitué des activités de conception, d'analyse et de déploiement. Afin de répondre au mieux aux objectifs du MDA, les activités d'analyse sont réalisées plut tôt dans le cycle en particulier au niveau conception ou PIM. Cette pratique permet de détecter rapidement les erreurs de développement, ce qui diminue considérablement le coût de traitement de ces erreurs et réduit le délai de mise sur le marché (*time-to-market*).Figure 2.5 illustre le principe de développement des systèmes temps réel embarqués dans un cadre MDA. En effet, le modèle de spécification, le CIM (Computation Independent Model) dans une approche MDA, traduit les exigences fonctionnelles et les contraintes temporelles du cahier des charges. A partir de ce modèle, l'activité de conception décrit une réalisation possible de l'application en introduisant des choix architecturaux (introduction de la concurrence, dépendance de données, etc.) mais tout en restant indépendant de la technologie logicielle(PIM). Une activité d'analyse permet de valider ces choix afin de vérifier si les contraintes temporelles sont respectées. Ensuite, la phase de déploiement permet de générer à partir de ce PIM, plusieurs modèles spécifiques à différentes plateformes logicielles(PSM).

Avec cette initiative sur le MDA, l'OMG a introduit le concept de plateforme, cependant aucune précision n'est donnée sur sa forme et sur sa prise en compte dans le processus de développement. Cela ouvre la possibilité de plusieurs interprétations de la notion de plateforme et de modèle de plateforme. De plus la façon de transformer un PIM en PSM en tenant compte de cette dernière n'est pas détaillée. En effet, lors du passage du PIM vers PSM à l'aide d'une transformation, le PDM peut être décrit soit implicitement dans la transformation ou dans l'application faisant ainsi une partie intégrante de la transformation ou de l'application, soit il est décrit explicitement dans des modèles à part, et il sera alors passé en entrée de la transformation, avec le PIM.

2.2.2.3 Caractérisation d'une plateforme dans un cadre IDM

Dans le cadre d'une ingénierie dirigée par les modèles, Atkinson et Kühne [17] définissent une plateforme, comme étant un système permettant la réalisation des objectifs d'une application. Par cette définition, les auteurs cherchent à montrer que l'exécution n'est pas le seul objectif d'une plateforme. En effet, au sens de l'IDM, une plateforme peut être utilisée pour des activités d'analyse, stockage de donnée ou même pour la représentation des modèles.

Dans ce même contexte, Almeida [13] introduit deux types de plateformes : plateforme abstraite et plateforme concrète. Une plateforme abstraite au sens d'Almeida est une plateforme fictive définissant un ensemble de concepts qui peuvent ne pas avoir une implantation réelle. Dans son approche, Almeida introduit aussi une plateforme abstraite comme étant une plateforme idéale de point de vue du développeur de l'application. Cette plateforme est définie pour une utilisation précise à une phase donnée du cycle de développement (généralement pour garantir l'indépendance par rapport à une plateforme d'exécution). CORTA [33] est un exemple de plateforme abstraite définit dans un cadre IDM pour la spécification des applications multi-tâches d'une façon indépendante des exécutifs temps réel. Contrairement à une plateforme abstraite, une plateforme concrète pour Almeida ne peut être qu'une plateforme d'exécution.

Selic [79] [78] définit une plateforme comme un ensemble de ressources qui fournit des services qui peuvent être accessibles par un ou plusieurs clients (des applications). Selic souligne ,à ce titre, qu'une plateforme au travers ses ressources et services doit être en mesure de satisfaire le besoin de l'application.

Face à cette diversité, l'OMG propose sa propre définition pour le concept de plateforme. Cette définition, dont la traduction est donnée ci-dessous, spécifie qu'une plateforme fournit un ensemble de fonctionnalités, au travers d'interfaces et des patrons d'utilisation de ces fonctionnalités.

« Une plateforme est un ensemble de sous-systèmes et de technologies qui fournissent un ensemble cohérent de fonctionnalités au travers d'interfaces et la spécification de patrons d'utilisation que chaque sous-système qui dépend de la plateforme peut utiliser sans être concerné par les détails et sur la façon dont les fonctionnalités sont implantées » [65]

En gros, une plateforme dans l'IDM est plus qu'une plateforme d'exécution. Elle est dite abstraite, si elle est définie pour une utilisation prévue, autre que l'exécution telle que l'analyse ou la simulation, à un instant donné du cycle de développement. Sinon, elle est dite concrète.

2.2.3 Synthèse

Nous avons évoqué dans cette première partie les points clés de cette étude dans le but cadrer le contexte de nos travaux. Nous avons ainsi introduit les systèmes temps réel embarqués. Plus précisément, nous avons fixé les hypothèses considérées dans cette étude pour

la conception de ces systèmes. **Nous considérons, dans cette thèse, des applications temps réel s'exécutant sur une architecture monoprocesseur et ayant comme support d'exécution logiciel un système d'exploitation temps réel (RTOS)**. Nous avons montré que, de part leur nature, ces systèmes requièrent une étape de vérification supplémentaire par rapport aux systèmes classiques en particulier une vérification des contraintes temporelles se basant sur **des techniques d'analyse d'ordonnançabilité**. Nous avons également présenté l'ingénierie dirigée par les modèles, en particulier, nous avons détaillé l'approche MDA et nous avons survolé des travaux qui ont abordés la notion de plateforme dans le cadre de l'IDM. Ces deux points constituent la base des contributions de cette thèse.

2.3 Les plateformes d'exécution

Il n'existe pas à nos jours une définition unique et précise du concept plateforme. Classiquement, une plateforme référence la structure matérielle permettant l'exécution du logiciel. Cette vision a rapidement évolué avec la montée en niveau d'abstraction et l'apparition de la structuration en couches des plateformes [57]. Cette représentation en couches successives d'infrastructure matérielle, de système d'exploitation, de machine virtuelle et d'intergiciel rend la distinction entre l'application et la plateforme fortement liée au point de vue où l'on se place. En effet, un système d'exploitation peut être vu comme une application du point de vue d'un fournisseur de support matériel, et comme une plateforme d'exécution du point de vue d'un développeur d'applications logicielles. Par conséquent, la caractérisation d'un système comme étant une plateforme ne dépend pas de la nature du système lui-même mais plutôt de l'utilisation prévue pour ce système dans un contexte donné.

Hypothèse. Dans cette étude, nous faisons une abstraction de l'architecture matérielle et nous nous intéressons uniquement *aux plateformes logicielles*. En outre, nous considérons les systèmes d'exploitation temps réel (*RTOS*) comme plateformes d'exécution pour les applications visées dans cette étude.

2.3.1 Représentations des plateformes d'exécution dans l'IDM

Dans un cycle de développement basé sur les modèles, une plateforme logicielle d'exécution peut être représentée dune façon enfouie, implicite ou explicite.

La représentation enfouie est généralement utilisée dans les approches qui visent la génération du code d'une application à travers les transformations de type modèles en texte. Dans ce cas, les caractéristiques de la plateforme sont capturées dans la transformation. On peut citer dans ce contexte l'outil OCARINA [45] qui permet la génération de code Ada ou C pour l'intergiciel PolyORB II [91] à partir d'une description AADL [68] de l'application.

La représentation implicite est basée sur la définition d'un méta-modèle de la plateforme considérée [74] [28] [51]. Ce méta-modèle est implémenté sous forme d'un profil UML ou d'un DSL (Description Specification Language) et est utilisé pour la description du modèle applicatif. Dans ce cas, la plateforme est représentée de façon implicite dans l'application.

Dans une représentation explicite, la plateforme est décrite dans un modèle indépendant de la transformation et de l'application. Dans ce cas, la génération des modèles applicatifs spécifiques à une plateforme se traduit par une transformation du type modèle en modèle ayant comme entrées : le modèle de l'application indépendant de toute plateforme et le modèle de plateforme.

Hypothèse. Afin de répondre aux principes de le MDA, dans ce travail, nous adoptons *une représentation explicite* des plateformes d'exécution (RTOS).

Cette dernière hypothèse exige alors la définition des modèles de plateformes d'exécution (en d'autres termes des modèles des RTOS), ce qui nécessite l'utilisation d'un langage approprié adapté à nos besoins de modélisation. Cette problématique est l'objet de la partie suivante.

2.3.2 Modélisation des plateformes d'exécution

Selon [61], un modèle de plateforme est une représentation abstraite mais suffisamment précise d'une plateforme d'exécution. Un modèle de plateforme doit être focalisé sur les aspects de celle-ci nécessaires pour son utilisation. En effet, une plateforme d'exécution logicielle peut être modélisée par exemple dans le but de conception d'une application (description d'une application multi-tâches), d'analyse ou de génération de code. La précision du modèle de cette plateforme dépend fortement de l'objectif pour lequel ce modèle a été crée.

Hypothèse. Dans cette thèse, nous nous intéressons à la modélisation des plateformes d'exécution logicielles (RTOS) dans le but de permettre le déploiement d'une application sur un RTOS en assurant le respect des contraintes temporelles. Plus précisément, les modèles des plateformes considérés dans cette étude se limitent aux concepts des RTOS nécessaires pour la vérification temporelles en utilisant les techniques d'analyse d'ordonnançabilité.

2.3.2.1 Modèle de plateforme pour l'analyse d'ordonnançabilité

Dans le contexte de modélisation des plateformes logicielles, Marvie et al. [73] proposent de modéliser une plateforme selon trois vues distinctes : (a) une vue statique des ressources fournies par la plateforme (telles que les tâches logicielles pour un système d'exploitation multi-tâches), (b) une vue dynamique des traitements réalisés par la plateforme (telles que la création et l'activation d'une tâche) et (c) une vue qui traite le cycle de vie des ressources sous l'influence des traitements qui leurs sont appliqués (le cycle de vie d'une tâche est, par exemple, borné par sa création et sa destruction). Dans la même vision, Thomas [86] identifie quatre axes pour modéliser une plateforme logicielle d'exécution : (1) la caractérisation des concepts d'une plateforme (les ressources d'une plateforme)(2) la caractérisation des traitements (les services associées aux ressources).(3) la caractérisation des règles d'utilisation d'une plateforme (des contraintes et des patrons d'utilisation des ressources et des services offertes par une plateforme).(4) La caractérisation de comportement observable d'une plate-

forme (le cycle de vie d'une ressource sous l'influence des services).

Un modèle de plateforme logicielle pour l'analyse d'ordonnançabilité doit être focalisé sur les aspects de celle-ci nécessaires pour vérifier les contraintes de temps d'une application. En d'autres termes, ce modèle décrit uniquement les éléments de la plateforme qui interviennent pour la vérification temporelle. Citons quelques concepts d'une plateforme logicielle d'exécution nécessaires pour la vérification temporelles des applications visées dans cette étude :

- **Entité ordonnançable(Tâche)** : Toute entité qui implémente les fonctions applicatives et qui est gérée par un ordonnanceur. L'ordre des priorités des tâches ainsi que leur type (périodique, sporadique, etc.) constituent les aspects de ce concept qui ont une influence direct sur les résultats d'analyse. Si le concept d'une tâche périodique n'existe pas par défaut dans un RTOS, le modèle de plateforme introduit un patron pour la description de ce dernier.
- **Ordonnanceur** : C'est un concept du noyau qui détermine la prochaine tâche à exécuter selon une politique d'ordonnancement. Cette politique ainsi que le mode de l'ordonnanceur (préemtif ou non préemptif) constituent l'ensemble des propriétés qui caractérisent ce concept et qui ont un impact sur les résultats d'analyse.
- **Ressource Partagée** : C'est une ressource qui implémente une section critique. Cette ressource est en général protégée par un mécanisme et une politique de la file associée qui gère l'accès simultané à cette dernière (Sémaphore ou Mutex). De plus, cette ressource peut être caractérisée par un protocole de synchronisation afin d'éviter les problèmes d'inversion de priorité et inter-blocage détaillés dans 5.3.2. Le type de ce protocole a une influence sur les résultats d'analyse.

Il est clair alors qu'une description structurelle est suffisante pour l'élaboration des modèles de plateformes pour l'analyse d'ordonnançabilité. Ainsi, dans cette étude, Nous nous focalisons sur le premier et le troisième axes identifiés par Thomas pour la création des modèles de plateformes logicielles d'exécution. Voyons maintenant quelques exemples de langage de modélisation de ces plateformes.

2.3.2.2 Langages de modélisation des plateformes

Plusieurs travaux se sont intéressés à la définition de langages pour la description des plateformes pour l'exécution de logiciel. Ces langages peuvent être basés sur le langage UML, les profils UML, ou des langages spécifiques de type DSL (Domain Specific Language).

AADL

AADL (Architecture Analysis and Design Language) [68], standardisé par le SAE (Society of Automotive Engineers) en 2004, est un langage de description d'architectures pour les systèmes temps réel embarqués. L'élément de base du langage AADL est le composant. Chaque composant est défini par son type et son implémentation (décrit le contenu du composant en termes de sous-composants, code, etc.). Les composants sont liés à travers des

ports et des connexions. Un port est un point d'entrée et/ou de sortie d'un composant, une connexion permet de relier deux ports. Un type de composant AADL appartient à une catégorie. Ces catégories sont prédéfinies et se décomposent en catégories matérielles, catégories logicielles ou catégories système. . Les quatre catégories de base pour le logiciel permettent la description d'une plateforme avec les concepts suivants : Thread (unité d'exécution concurrente qui peut être ordonnançable), Data (structure de données qui peut être partagée par les composants), Thread group (permet de créer une hiérarchie dans les threads) et Process (un espace mémoire pour l'exécution des threads). À chaque composant on peut associer des propriétés et leur donner des valeurs. Celles-ci permettent de caractériser le composant. Certaines propriétés sont prédéfinies, c'est-à-dire qu'elles sont identifiées par un nom, un type et la liste des catégories de composants sur lesquelles elles s'appliquent. Par exemple les tâches (thread) disposent de propriétés temps-réel telles que la période, l'échéance ou la durée d'exécution. De nouvelles propriétés, et de nouveaux types de propriétés, peuvent aussi être définis par l'utilisateur et associés à tout ou partie des catégories de composants. Ce mécanisme de propriétés est un point fort d'AADL en matière d'extensibilité. Grâce à lui, toute notion spécifique au besoin de l'utilisateur peut être prise en compte dans sa description.

Pour la modélisation des plateformes, le langage AADL ne peut pas être vu comme étant un méta-modèle de plateforme. En effet, la sémantique d'exécution des composants logiciels est clairement spécifiée dans la norme. Par contre, ce langage peut être considéré comme une plateforme abstraite. En effet, AADL permet de décrire explicitement des applications logicielles. Il est également possible de générer des modèles spécifiques à des plateformes d'exécution concrètes à partir d'une description AADL. Par exemple, dans [32] les auteurs proposent un environnement pour la génération des systèmes ARINC653 [18] à partir d'une description AADL.

Metropolis

En se basant sur l'approche Platform Based Design [75], le projet Metropolis [72] offre un cadre méthodologique pour le développement des systèmes embarqués. Le point clé de ce projet est la définition d'un méta-modèle permettant de représenter l'application, l'architecture et le déploiement de l'application sur l'architecture. Ce méta-modèle est définit avec une sémantique précise permettant ainsi à l'environnement Metropolis de couvrir presque toutes les phases de cycle de développement : conception, simulation, analyse formelle et synthèse. Pour modéliser une plateforme (matérielle ou logicielle), le méta-modèle Metropolis définit trois concepts de base : les unités d'exécution concurrente (*Process*), les *Media* qui sont utilisées pour connecter les interfaces des Process et assurer ainsi la communication des données et la synchronisation entre eux et le *Quantity manager* permettant de contrôler l'accès aux Media. Toutes les instances de ces concepts sont regroupées en Netlist. L'environnement Metropolis offre la possibilité d'exprimer, à un haut niveau, des contraintes. Ainsi, il intègre en complément des langages formels, tels que LTL (Linear Temporal Logic) ou LOC

(Logic Of Constraints) permettant la vérification de ces contraintes.

Le méta-modèle introduit dans le cadre du projet Metropolis est un méta-modèle de plateforme permettant ainsi une description explicite d'une plateforme. Cependant, la généricité du méta-modèle de cette approche, ne facilite pas la description des plateformes dans un domaine particulier [31]. En effet, les concepts de *Process* et de *Media* sont utilisés aussi bien pour la description de l'application que pour la description de la plateforme. Ceci limite l'utilisation de cette approche à des fins de modélisation détaillée des ressources d'une plateforme logicielle.

TUT-Profile

TUT-profile [51] est un langage basé sur l'extension de la version 2.0 d'UML. Ce profil a été proposé initialement pour la conception des systèmes temps réel embarqués et la description de l'architecture matérielle. Il a été étendu, par la suite, pour supporter la modélisation des plateformes logicielles d'exécution [15]. Pour ce faire, TUT-profile propose quatre concepts : *SwPlatform*, *SwPlatformLibrary*, *SwPlatformComponent* et *SwPlatformProcess*. Le premier concept, *SwPlatform*, caractérise une plateforme logicielle dans sa globalité. Le *SwPlatformLibrary* permet de décrire un package regroupant les concepts d'une plateforme. Ainsi, une plateforme logicielle est constituée d'un ensemble de librairies logicielles. Chaque concept de la plateforme logicielle est représenté par une classe et caractérisé comme étant un *SwPlatformComponent*. Finalement, le *SwPlatformProcess* est représenté par un élément typé dont le type correspond à un concept de la plateforme annoté par *SwPlatformComponent*.

Bien que ce langage présente l'avantage d'étendre un standard (UML), il reste encore trop générique pour la modélisation détaillée des ressources d'une plateforme logicielle. En effet, toutes les ressources offertes par la plateforme sont annotées par *SwPlatformComponent*. Ainsi, une tâche, une ressource ou un ordonnanceur sont tous modélisés de la même façon dans ce profil. Ceci présente une limitation pour ce profil pour garantir la spécification d'un modèle décrivant les ressources d'une plateforme avec une sémantique précise.

UML-MARTE

Le profil MARTE (UML profile for Modeling and Analysis of Real Time and Embedded systems) [64], standardisé par l'OMG en août 2007, permet de modéliser des plateformes d'exécution matérielles ou logicielles à différents niveaux d'abstraction. Ce profil a remplacé le profil SPT (Schedulability, Performance and Time) [66] en assurant la couverture de tous les aspects logiciels et matériels des systèmes embarqués temps réel. L'objet de ce profil est de soutenir par une riche base d'annotations facilitant l'utilisation d'UML pour la modélisation et l'analyse des systèmes temps réels embarqués. Le profil MARTE est organisé en trois paquetages principaux : « MARTE Foundations », « MARTE Design Model » et « MARTE Analysis Model ». Le premier paquetage fournit les bases du langage traitant la modélisation des propriétés non-fonctionnelles du temps, des ressources génériques et des allocations

des systèmes embarqués temps réel. Les deux autres paquetages sont des spécialisations du premier se focalisant sur un aspect particulier : Le paquetage « MARTE Design Model » fournit les artéfacts nécessaires pour modéliser les applications et les plates-formes d'exécution matérielles ou logicielles à différents niveaux d'abstraction. Quant au paquetage « MARTE Analysis Model », il définit les concepts nécessaires pour analyser les systèmes. En particulier pour la modélisation des plateformes logicielles, MARTE offre un sous profil défini dans le paquetage « MARTE Design Model » qui est SRM (Software Resource Modeling). Ce profil spécialise le sous profil GRM (Generic Resource Modeling) appartenant au paquetage « MARTE Foundations ». SRM définit les artéfacts permettant de décrire finement trois familles de ressources logicielles : les ressources d'exécution concurrentes (comme une interruption ou une tâche), les ressources d'interaction entre les entités concurrentes (comme une boîte aux lettres ou les mécanismes de sémaphore) et les ressources de gestion des entités matérielles et logicielles (comme les ordonnanceurs). Le profil SRM est organisé en quatre paquetages comme le montre la figure 2.6.

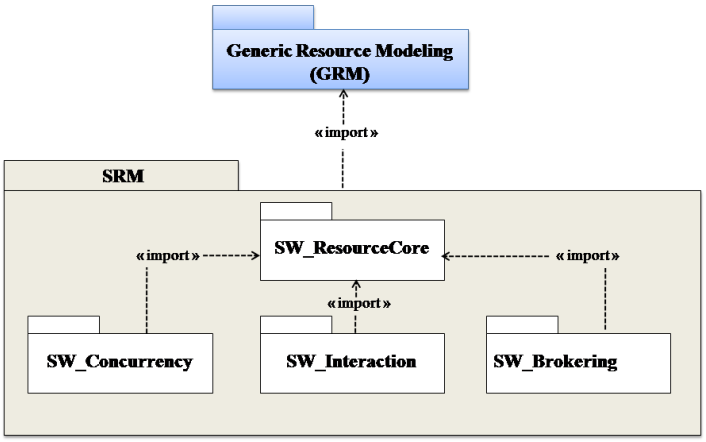


FIGURE 2.6 – *Structure du profil SRM*

Le premier paquetage « SW_ResourceCore » fournit les ressources de conception de base du logiciel. En particulier ce paquetage définit le concept SwResource qui référence toutes les ressources d'une plateforme logicielle. Ce concept est caractérisé par les propriétés en commun des ressources logicielles tel que l'identificateur identifierElements de la ressource par exemple qui permet de la référencer. Le paquetage « SW_Concurrency » fournit les artefacts de modélisation pour décrire des contextes d'exécution logicielle concurrente. Dans SRM, il existe deux types de ressources concurrentes : les ressources ordonnançables et les interruptions. Ainsi, le stéréotype SwSchedulableResource est créé pour identifier le premier cas et le stéréotype InterruptResource pour caractériser le second cas. Le paquetage « SW_Interaction » offre des stéréotypes qui décrivent les mécanismes de communication et de synchronisation des plates-formes logicielles d'exécution. Les mécanismes de communication peuvent être de deux types : communication par échange de messages ou par accès à une variable partagée. Chaque type de communication est caractérisé par un stéréotype dans

SRM qui sont MessageComResource et SharedDataComResource respectivement. D'autre part, le stéréotype SwMutualExlusionResource décrit les ressources d'exclusion mutuelle synchronisant l'accès concurrent à une même variable partagée. Chaque stéréotype peut avoir une ou plusieurs propriétés permettant ainsi une caractérisation plus fine du concept. Enfin, le paquetage « SW_Brokering » permet la modélisation de la gestion des ressources logicielles et matérielles.

Le profil SRM se base sur un métamodèle qui propose des concepts métiers concrets pour la description des plateformes logicielles d'exécution. De part le nombre important des artefacts qu'il offre et de la capacité de configuration de ces artefacts, SRM peut être un bon profil d'annotation pour une description détaillée et explicite des ressources offertes par une plateforme logicielle.

RTEPML

RTEPML (Real-time Embedded Platform Modeling Language) [29] est un langage de modélisation des plates-formes logicielles d'exécution. Ce langage s'appuie sur les concepts définis par SRM d'UML MARTE. RTEPML est une implantation de SRM dans un contexte DSML. Ce langage permet une description détaillée des différentes ressources d'un exécutif temps réel. Un ensemble de transformations génériques ont été aussi proposé avec RTEPML afin de permettre l'utilisation de ce langage pour la description des plateformes logicielles ainsi que le déploiement des applications sur ces plateformes.

2.3.3 Synthèse

Le tableau 2.1 montre une comparaison entre les différents langages de modélisation des plateformes étudiés. Cette comparaison porte tout d'abord sur la capacité du langage à modéliser les ressources d'une plateforme logicielle d'exécution avec une sémantique assez précise permettant de capturer toutes les informations nécessaires pour la vérification temporelle. Nous nous limitons dans cette comparaison aux ressources liées à la concurrence, l'ordonnancement et la synchronisation. La comparaison porte aussi sur la capacité d'extension d'un langage (extensibilité) et sur sa capacité à assurer des transformations génériques permettant une capture non ambiguë des concepts de la plateforme. Finalement, pour chaque langage étudié, nous précisons s'il s'agit un méta-modèle de plateforme permettant une description explicite de la plateforme d'exécution ou d'un modèle de plateforme menant à des descriptions implicites de cette dernière.

Nous remarquons que tous les langages étudiés, sauf AADL, offrent des artéfacts permettant une modélisation explicite des plateformes d'exécution pour le logiciel. Ils sont ainsi considérés comme des méta-modèles de plateformes.

En termes de ressources, les langages de modélisation Metropolis et TUT-Profile proposent des concepts incomplets (-) ou abstraits (+) pour la modélisation des ressources d'une plateforme logicielle d'exécution pour des besoins d'analyse temporelle. Pour cela, les trans-

formations basées sur ces langages sont généralement ambiguës et peu génériques.

TABLE 2.1 – Comparaison des langages de modélisation des plateformes logicielles d'exécution

Approches	Ressources			Généricité des Transformations	Extensibilité	Méta-Modèle de plateformes
	Concurrence	Ordonnancement	Synchronisation			
AADL	++	++	++	-	+	×
Metropolis	+	+	+	-	-	✓
RTEPML	++	++	++	++	-	✓
TUT-Profile	-	-	-	-	++	✓
MARTE/SRM	++	++	++	++	++	✓

Le DSL RTPEML et le profil SRM de MARTE permettent une description des ressources d'une plateforme d'exécution avec une sémantique précise adaptée aux besoins de la vérification temporelle (++). Ainsi, ces langages permettent des transformations génériques basées sur une capture précise des différents concepts décrits dans les modèle des plateformes logicielles d'exécution.

L'IDM ne préconise pas l'utilisation d'un langage de modélisation en particulier. Cependant, depuis sa standardisation par l'OMG, UML s'est imposé comme standard de modélisation dans beaucoup de domaines. De plus, les langages basés sur UML possèdent de puissantes capacités d'extensions. Par conséquent, dans cette thèse, nous nous basons sur le profil SRM du standard MARTE pour la modélisation des plateformes logicielles d'exécution.

2.4 Plateformes cibles et processus de développement des STRE

L'objectif principal d'un processus de développement des systèmes temps réel embarqués (STRE) est de traduire les exigences du cahier des charges (fonctionnelles et non fonctionnelles) en une application déployée sur une plateforme cible. Dans un cadre IDM, ce

processus passe en général par trois activités, chaque activité est supportée par un type de modèle. Une activité de spécification permet de traduire les exigences et les contraintes d'une application en un modèle de spécification, une activité de conception permet d'introduire des choix architecturaux afin de décrire une réalisation possible de cette application indépendamment de toute plateforme dans un modèle de conception et enfin une activité de déploiement permet de rendre cette réalisation spécifique à une technologie logicielle dans un modèle d'implémentation. La plateforme d'exécution cible est prise en compte dans un tel processus lors du déploiement de la description architecturale de l'application sur une plateforme logicielle afin d'assurer son exécution.

Pour la prise en compte des contraintes temps réel, le processus de développement des STRE introduit, durant la phase de conception, une activité de vérification des propriétés temporelles. Afin de vérifier un système, il faut construire un modèle formel approprié permettant cette vérification. Un point clé de la construction de ce modèle porte sur l'introduction de la concurrence ainsi que la sémantique d'exécution permettant son interprétation. La définition d'une sémantique d'exécution des architectures pour des raisons d'analyse nécessite une abstraction des informations liées à la plateforme logicielle d'exécution telles que la politique d'ordonnancement, l'ordre de priorité, etc. Ceci nous amène à la nécessité de prendre en compte la plateforme logicielle pour la vérification des applications temps réel.

Dans cette partie, nous nous intéressons tout d'abord à quelques travaux qui se sont focalisés sur la prise en compte des aspects temps-réel lors du processus de développement des STRE. En particulier, nous nous arrêtons sur la façon de prendre en compte la plateforme logicielle pour vérifier les contraintes temporelles de l'application. Ensuite, nous présentons les différentes études qui se sont intéressées au déploiement d'une application sur une plateforme logicielle, avant de donner le bilan de cette partie et le positionnement par rapport à ce qui existe pour introduire la contribution de cette thèse.

2.4.1 Prise en compte des plateformes pour la vérification des contraintes temporelles des applications

Afin de permettre la vérification des contraintes temporelles lors d'un processus de développement des STRE, certains travaux se sont orientés vers une vérification des modèles spécifiques à des plateformes d'exécution logicielles. En effet, un modèle de l'application spécifique à une plateforme logicielle définit en général une description de la réalisation de cette l'application sur la plateforme considérée. Cette réalisation exprime la concurrence et introduit une sémantique d'exécution claire permettant ainsi la vérification des contraintes temporelles de l'application. Dans ce contexte, citons le travail de Moore, présenté dans [60], qui a proposé d'étendre RT-UML Profile (UML Profile on Scheduling, Performance and Time) pour supporter l'infrastructure OSEK [6]. Dans ce travail, la plateforme OSEK est représentée de façon implicite au niveau méta-modèle afin de permettre la description des applications. En outre, l'auteur propose d'ajouter au profil RT-UML un sous-profil personnalisé afin de faciliter l'analyse d'ordonnançabilité des systèmes OSEK.

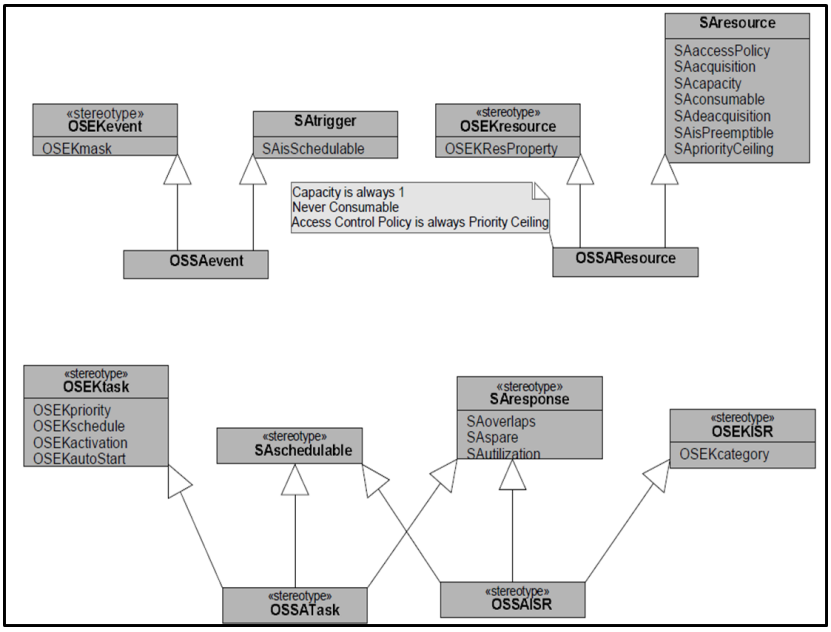


FIGURE 2.7 – *Extrait du sous-profil définit dans [60] pour l'analyse les systèmes OSEK*

La figure 2.7 donne un extrait du sous-profil OSEK pour l'analyse d'ordonnançabilité. Ce sous-profil définit les concepts nécessaires et leurs propriétés nécessaires pour vérifier qu'un système OSEK est ordonnançable. Dans [89], les auteurs proposent une approche basée sur les modèles pour la conception et la vérification des systèmes OSEK afin de faciliter le développement des systèmes automobiles. Contrairement au travail précédent, ce travail se base sur une plateforme abstraite appelée SmartOSEK [94], compatible avec le standard OSEK, décrite explicitement dans un modèle. Afin de générer un modèle de l'application spécifique à OSEK, les auteurs suivent une approche MDA ; partant d'une application décrite avec UML et indépendante de toute plateforme et du modèle de la plateforme SmartOSEK, les auteurs génèrent le modèle de l'application spécifique à OSEK par la définition de règles de transformation entre les concepts UML et SmartOSEK. Ainsi, l'étape de vérification des propriétés temporelles de l'application est réalisée sur ce modèle généré spécifique à la plateforme OSEK par l'intermédiaire d'un simulateur (SmartOSEK Simulator) fournit avec la méthodologie. Dans [21], les auteurs introduisent une approche pour la vérification des applications spécifiques à l'API Real-Time Java (RT-Java) [25]. Contrairement aux deux approches précédemment cités, les auteurs n'optent pas pour une vérification temporelle des modèles spécifiques à RT-java, mais plutôt cherchent à assurer la traçabilité des contraintes temps réel à partir du modèle de l'application vers le code. Cette approche utilise le profile RT-UML pour annoter les modèles des applications avec les propriétés temporelles. Ensuite, elle définit des règles de mapping entre les stéréotypes et les tagged values du profil RT-UML d'une part et les concepts RT-Java d'autre part afin de propager les contraintes temps réel au niveau code. Ainsi, une analyse du code RT-Java de l'application est effectuée dans le but de vérifier si les contraintes de temps déjà propagées sont respectées. La plateforme

d'analyse est représentée dans cette approche d'une façon enfouie dans les transformations permettant de générer un code analysable.

D'autres travaux s'orientent vers une vérification temporelle, durant la phase de conception, des modèles des applications indépendants de toute plateforme d'exécution logicielle. L'objectif de ces approches en général est la description d'une application dans un modèle d'architecture qui respecte les contraintes de temps. Pour cela, ces approches font une abstraction de la plateforme matérielle (en supposant que celle-ci est fixe) et considèrent des hypothèses sur la plateforme logicielle pour rendre le modèle d'architecture analysable et garder en même temps l'indépendance vis-à-vis de l'implémentation. Dans ce cas, les hypothèses à considérer dépendent fortement de la technique d'analyse à appliquer. Parmi ces approches, notons le travail décrit dans [19], qui montre une approche pour la synthèse des modèles d'architecture à partir des blocs fonctionnels décrivant une application temps réel. L'objectif des auteurs, dans ce papier, est de générer un modèle d'architecture qui satisfait les contraintes de temps. Pour cela, ils proposent d'effectuer un test d'ordonnançabilité basé sur l'utilisation du processeur [20]. Pour cette technique d'analyse, seule une hypothèse liée à la politique d'ordonnancement est nécessaire. Ainsi, dans ce papier, les auteurs supposent que l'ordonnanceur est de type EDF (Earliest Deadline First)[93]. Dans [49][62] les auteurs cherchent aussi à générer un modèle d'architecture d'une application temps réel qui satisfait les contraintes de temps. Pour cela, ces deux travaux utilisent une technique d'analyse basée sur le calcul d'une borne supérieure sur les temps de réponse des tâches constituant le modèle. Ainsi, afin de permettre ce type d'analyse, les auteurs dans [49] supposent qu'ils disposent d'un ordonnanceur préemptif de type priorité fixe [52]. Ils supposent aussi que l'ordre de priorité est croissant c'est-à-dire plus la valeur de priorité est grande, plus la tâche est prioritaire. Pour le calcul du temps de blocage dû à la synchronisation des tâches et qui est un facteur important dans le calcul du temps de réponse, ce travail donne une estimation de ce temps. En effet, selon l'application, les auteurs considèrent une hypothèse sur la valeur du temps blocage. Dans la méthodologie Optimum proposée dans [62], les auteurs supposent aussi qu'ils disposent d'un ordonnanceur préemptif de type priorité fixe [52], cependant, l'ordre de priorité est décroissant dans cette approche. Pour le temps de blocage, les auteurs dans ce travail cherche à avoir une valeur plus précise que le travail précédent en essayant de calculer sa valeur. Ceci nécessite une hypothèse supplémentaire sur la plateforme logicielle et qui correspond au protocole de synchronisation utilisé. Ce protocole est fixé à Priority Ceiling Protocol (PCP) [43] dans le cadre de cette approche.

Différentes approches sont proposées dans le but de vérifier les contraintes temporelles d'une application dans un cycle de développement des systèmes temps réel embarqués. Cette vérification nécessite une prise en compte de quelques aspects de la plateforme logicielle d'exécution. Pour cela, ces travaux s'orientent soit vers une vérification des modèles des applications spécifiques à une plateforme d'exécution particulière [60][89] [21], soit vers une considération des hypothèses sur la plateforme cible dans le but de rester indépendant de cette dernière et suivre la ligne de le MDA [19][49][62].Dans la partie suivante, nous allons introduire quelques travaux qui ont été proposées dans un cadre IDM dans le but de

déployer une application sur une plateforme logicielle.

2.4.2 Prise en compte des plateformes lors du déploiement des applications

Le déploiement d'une application sur une plateforme technologique consiste à générer un modèle de l'application spécifique à cette technologie. Ce modèle représente l'implémentation de l'application sur la plateforme considérée à un niveau plus abstrait que le code. L'approche IDM, telle que préconisée par le MDA, a fait une séparation claire entre les aspects métiers et les aspects liés à la plateforme en les décrivant dans des modèles distincts. Le point fort d'une telle séparation offre la possibilité de générer plusieurs modèles spécifiques à différentes plateformes (PSM) à partir du même modèle de l'application indépendant de la plateforme (PIM). C'est dans ce cadre, que plusieurs approches ont été proposées dans le but d'automatiser ce déploiement en proposant des processus basés sur une prise en compte explicite de la plateforme logicielle cible. Parmi ces approches, citons le travail présenté dans [86]. Ce travail définit un cadre méthodologique pour le portage d'une application d'une plateforme à une autre en se basant sur une description explicite et commune des plateformes source et cible. Cette description est réalisée en utilisant le profil SRM comme un langage pivot pour l'élaboration d'une infrastructure de transformation indépendante des plateformes. Afin de permettre le déploiement d'une application sur une plateforme logicielle, le sous profil HLAM (High Level Application Modeling) de la norme MARTE [64] est utilisé pour décrire initialement cette dernière. Ce profil autorise la description des exécutions concurrentes à un haut niveau d'abstraction et peut être considérée comme une plateforme logicielle. Ainsi, le déploiement correspond à une opération de portage de l'application de la plateforme HLAM décrite en utilisant SRM à une autre plateforme explicitement décrite en utilisant aussi SRM. En effet, pour chaque instance du modèle de départ dont le type est stéréotypé dans la plateforme source (la plate-forme de départ qui est ici HLAM), s'il existe une ressource stéréotypée de la même façon dans la plate-forme cible (la nouvelle plate-forme sur laquelle l'application doit être déployée), cette ressource est instanciée.

Dans la même idée, le travail proposé dans [29], vise à mettre en place un environnement de déploiement d'une application sur différentes plateformes logicielles décrite en utilisant le langage RTEPML. Dans ce travail, l'auteur a également défini un ensemble de règles de transformation génériques pour automatiser le déploiement des systèmes multi-tâches. En effet, ce déploiement repose sur le principe d'intégration d'une application, décrite dans un méta-modèle de l'application, sur une plateforme explicitement décrite avec RTEPML. Ce mécanisme d'intégration consiste à générer un modèle de la plateforme enrichi des instances de ressources qui mettent en œuvre l'application. L'application est alors modélisée, ou intégrée dans le modèle de la plate-forme, au travers des instances de ressources mises à contribution pour son exécution.

Le travail présenté dans [30] complète le travail présenté dans [86] en tenant compte de l'aspect comportemental lors de la description des plateformes logicielles. L'objectif de

ce travail est d'automatiser le processus de déploiement d'une application multi-tâches sur une plateforme logicielle dans le but de générer un modèle détaillé de l'application spécifique à la plateforme permettant ainsi la génération d'un code complet. Pour cela, cette approche propose des heuristiques de modélisation permettant d'offrir des modèles détaillés des plateformes logicielles d'exécution. Ces modèles détaillés sont décrits d'une manière commune par le profil SRM du standard MARTE [64]. D'autre part, cette approche définit des transformations de modèles génériques basées sur la notion des patrons de conceptions comportementaux permettant la gestion des mécanismes de communication et de synchronisation d'une manière indépendante de la plateforme. L'utilisation de ces patrons permet une génération automatique des aspects comportementaux liés à la plateforme dans le modèle d'application spécifique à celle-ci.

Les travaux présentés dans cette partie s'intéressent plus au déploiement des applications sur une plateforme d'exécution logicielle qu'aux aspects temps réel. Leur objectif est de proposer des transformations génériques permettant le déploiement d'une application sur une plateforme en suivant les principes de le MDA.

2.4.3 Synthèse et discussion

Dans ce chapitre, nous avons présenté des travaux pour le développement des systèmes temps réel embarqués. Le point clé des travaux présentés est qu'ils introduisent implicitement ou explicitement la notion de plateforme logicielle dans leur étude. Nous avons ainsi groupé ces travaux selon l'objectif pour lequel la plateforme logicielle a été introduite.

Le premier groupe concerne les approches qui se sont focalisées sur l'aspect temps réel. Ces approches font appel à une étape de vérification des contraintes de temps qui nécessite à son tour une prise en compte de certaines propriétés de la plateforme logicielle. Afin d'introduire l'aspect plateforme pour des fins d'analyse, nous avons identifié deux solutions : La première solution consiste à vérifier des modèles spécifiques à des plateformes logicielles [60][89] [21]. Quant à la deuxième, elle s'oriente vers une vérification des modèles indépendants des plateformes en considérant des hypothèses sur ces derniers [19][49][62]. La première solution est applicable dans le cas où une seule plateforme est visée. Néanmoins, elle est peu bénéfique dans un contexte multiplateforme. Dans un tel scénario, la vérification des contraintes de temps se fait après le déploiement ce qui contredit en quelque sorte les principes du MDA et diminue la portabilité de l'application. La deuxième solution est avantageuse puisqu'elle permet une vérification des contraintes de temps avant le déploiement sur une plateforme logicielle c'est-à-dire durant la phase de conception. Cette solution est plus adaptée au contexte MDA puisqu'elle permet une vérification des contraintes de temps plus tôt dans le cycle d'une part, d'autre part, elle cherche à assurer la portabilité de l'application validée. Cependant, toutes les approches qui ont adoptées cette solution ne sont pas parvenu, dans leur processus, jusqu'au déploiement de l'application validée sur une plateforme d'exécution logicielle. En effet, le déploiement de l'application sur la plateforme considérée est une tâche non triviale. Les hypothèses logicielles considérées pour vérifier les

contraintes de temps de l'application durant la phase de conception peuvent être non vérifiées pour la plateforme logicielle cible sur laquelle l'application vise à être déployée. Ainsi, le modèle de l'application spécifique à cette plateforme peut ne pas correspondre au modèle de conception initial de l'application (différentes interprétions sémantiques des deux modèles) ce qui peut affecter les propriétés temporelles déjà vérifiées. Dans un tel scénario, le concepteur est obligé de reboucler pour revoir ses choix de conception. Cette opération peut se répéter plusieurs fois jusqu'à ce que l'origine du problème soit détectée. Ceci entraîne l'allongement du cycle de développement et contredit une des contraintes de conception actuelle qui est la réduction du délai de mise sur le marché (time-to-market).

Le deuxième groupe concerne les approches qui ont été proposées dans un cadre IDM dans le but de déployer une application sur une plateforme logicielle. Le point commun de ces travaux c'est de respecter le principe du MDA en faisant une séparation entre l'aspect applicatif et l'aspect plateforme. Ceci garantit la portabilité de l'application et la possibilité de déployer cette dernière sur différentes plateformes. Néanmoins, toutes ces approches [86][29][30] considèrent des applications temps réel non critiques, et portent moins d'attention, que les approches du groupe précédent, à l'aspect temps réel. Cependant, toutes ces approches considèrent des applications multi-tâches en introduisant la notion de concurrence et en particulier le concept de priorité. Seulement, aucune de ces approches n'a traité le problème de la sémantique liée à la priorité qui peut varier d'une plateforme à une autre. Il reste tout de même nécessaire de préserver les propriétés de l'application lors du passage des phases de conception aux phases d'implémentation. D'autre part, dans [86], l'auteur propose un processus génératif pour le portage d'une application d'une plateforme à une autre. Ce travail suppose que ce portage est réalisable quel que soit la plateforme, ce qui n'est pas toujours vrai dû à la diversité des plateformes qui peuvent ne pas être caractérisées de la même façon.

Le tableau 2.2 résume cette partie en présentant une comparaison des principales approches étudiées selon trois critères : l'aspect temps réel (les applications considérées par l'approche sont critiques), le déploiement (l'approche arrive jusqu'à la génération d'un modèle de l'application spécifique à une plateforme) et la portabilité (l'approche propose une solution adaptée à un contexte multiplateforme).

Par rapport aux travaux existants, cette thèse vise à satisfaire les trois critères mentionnés dans le tableau. Nous cherchons donc à permettre le déploiement d'une application temps réel critique sur différentes plateformes logicielles. Le processus que nous allons proposer doit garantir la portabilité de l'application en effectuant une vérification des contraintes temporelles durant la phase de conception. Notre contribution va tirer profit des travaux existants en essayant d'exploiter les résultats des approches qui ont traité le problème de vérification d'une application indépendante de toute plateforme logicielle, en particulier, l'approche Optimum [62]. L'objectif est d'étendre ces travaux dans le but de déployer cette application sur différentes plateformes logicielles en se basant sur les travaux qui ont été proposés dans ce contexte, en particulier, le principe de portage proposé par Thomas [86]. Cette thèse vise ainsi à définir un cadre méthodologique pour le concepteur dans le but de

TABLE 2.2 – Comparaison des langages de modélisation des plateformes logicielles d'exécution

(+) :Supporté(-) :Non Supporté	**Critères de comparaison**		
Les approches	**Aspect Temps-Réel**	**Déploiement**	**Portabilité**
A. Moore	+	+	-
L.B. Becker et al.	+	+	-
C. Bartolini et al.	+	-	+
Optimum	+	-	+
F. Thomas	-	+	+
M. Brun	-	+	+

le guider à déployer son application sur n'importe quelle plateforme logicielle tout en assurant à moindre coût le respect des contraintes de temps à l'implémentation. Nous allons montrer qu'une telle approche permet de détecter au plus tôt les problèmes de déploiement des applications critiques, ce qui contribue à la réduction des cycles de développement.

2.5 Conclusion

Dans ce chapitre, nous avons présenté le contexte général de nos travaux et au fur à mesure nous avons fixé les hypothèses considérées dans cette thèse. Nous avons également abordé la problématique de modélisation des plateformes logicielles d'exécution, qui sont des RTOS dans notre cas, à travers une définition d'un modèle de plateforme et une étude comparative de certains langages proposés dans la littérature pour créer ce dernier. Par ailleurs, nous avons étudié dans ce chapitre, la prise en compte d'une plateforme logicielle d'exécution à des fins d'analyse ou de déploiement. Le travail de synthèse des travaux existants nous a permis de conclure que le problème de déploiement multiplateforme des applications critiques temps réel n'a pas été traité.

Cette thèse vise alors à proposer des solutions, pour cette problématique, conforme au principe du MDA et qui permet le déploiement d'une application temps réel critique sur différentes plateformes d'exécution logicielles (RTOS). Tout d'abord, nous proposons de considérer explicitement la plateforme logicielle utilisée pour l'analyse au niveau conception. Cette contribution permettra de garantir l'indépendance par rapport au RTOS cible d'une part, et doit préparer la phase de déploiement de l'application temps réel sur ce dernier, d'autre part. Ensuite, en nous basant sur cette première contribution, nous introduisons un processus basé sur les modèles que nous appelons DRIM (Design Refinement toward Implementation Methodology). Ce processus permet de guider le déploiement multipltaforme des applications temps réel critiques. Nous définissons également une méthodologie pour la

description des modèles dans DRIM et nous proposons une mise en œuvre de ce processus. Ces différentes contributions sont les sujets de la suite de ce rapport.

Chapitre 3

DRIM : un processus pour le déploiement multiplateforme des applications temps réel

Dans ce chapitre, nous donnons un aperçu global des contributions de cette thèse. Nous commençons par expliquer notre contribution pour une prise en compte explicite de la plateforme logicielle, dite abstraite, utilisée pour l'analyse d'ordonnançabilité au niveau conception. Ensuite, nous présentons un processus que nous appelons DRIM (Design Refinement toward Implementation Methodology). Ce processus définit un ensemble d'étapes intermédiaires entre la phase de conception et la phase d'implémentation du flot IDM permettant ainsi le déploiement d'une application temps réel sur différents RTOS.

3.1 Introduction

Le chapitre précédent a montré que la vérification des propriétés temporelles d'une application temps réel, lors d'un processus conforme aux principes du MDA, nécessite une prise en compte de certains aspects de la plateforme logicielle. Une analyse de l'état de l'art a aussi montré que cette prise en compte, qui se traduit par des hypothèses intégrées implicitement dans l'application, peut conduire à des problèmes lors du déploiement de l'application temps réel sur le système d'exploitation temps réel cible. Afin de faire face à ces problèmes, nous proposons dans ce chapitre un processus [70] que nous appelons DRIM (Design Refinement toward Implementation Methodology) qui permet de guider le déploiement multiplateforme d'une application temps réel critique dans un contexte MDA.

Ce chapitre est structuré comme suit. La première partie explique notre contribution pour une prise en compte explicite de la plateforme logicielle afin de permettre la vérification temporelle et préparer par la suite le déploiement de l'application sur un RTOS. Ensuite, nous décrivons le processus DRIM proposé et nous expliquons ses différentes étapes.

3.2 Contribution à la prise en compte explicite des plateformes logicielles pour l'analyse

Dans [29], l'auteur compare différentes approches pour la prise en compte d'une plateforme dans le but de générer un modèle de l'application déployée sur une plateforme d'exécution (RTOS). Cette comparaison montre qu'une description implicite de la plateforme semble intéressante et facile à mettre en œuvre si une ou peu de plateformes cibles sont envisagées. Cependant, dans un contexte multiplateforme (plusieurs plateformes cibles sont envisagées), une description explicite de la plateforme d'exécution cible est plus adaptée. En effet, selon cette étude, une prise en compte explicite des plateformes permet de garantir des transformations plus génériques en assurant la réutilisation des modèles et la séparation des préoccupations.

Partant de ce constat, nous adoptons dans cette étude une description explicite des plateformes logicielles à des fins d'analyse. Pour cela, nous caractérisons tout d'abord les différents niveaux de plateformes de point de vue analyse dans un flot basé modèles. Ensuite, nous identifions les besoins pour la définition d'une plateforme abstraire pour l'analyse d'ordonnnaçabilité.

3.2.1 Modèles de plateformes de point de vue analyse dans l'IDM

Dans un flot typique IDM, la vérification des propriétés temporelles qui se base sur des techniques statiques d'analyse d'une application temps réel peut être réalisée à différents niveaux d'abstraction. Selon la technique d'analyse, la vérification des contraintes temps réel possède la particularité d'être étroitement liée à la plateforme d'exécution matérielle et/ou logicielle. Dans un cadre IDM, une plateforme peut être définie pour jouer un rôle précis

à n'importe quel stade du cycle de développement. Chaque utilisation prévue de la plateforme nécessite une description adaptée de cette dernière en occultant les détails inutiles. En particulier, comme illustré dans la Figure 3.1, pour des activités d'analyse un modèle de plateforme approprié doit être considéré à chaque niveau du flot IDM afin de permettre la vérification des contraintes de temps à ce niveau.

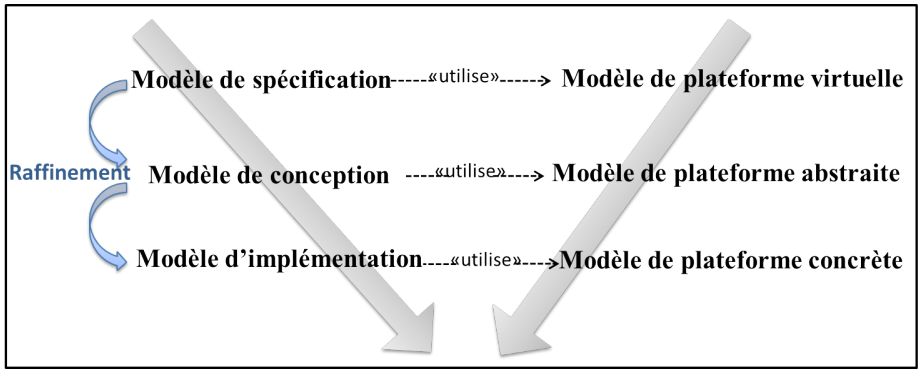


FIGURE 3.1 – *Les niveaux de plateformes de point de vue analyse dans un flot IDM*

Au niveau spécification, les techniques d'analyse comme les techniques d'allocation des budgets de temps aux différentes fonctions de l'application (*time budgeting* en anglais)[38] considèrent que la plateforme d'exécution est une plateforme idéale. A ce niveau, aucune limitation des ressources matérielles, ni logicielles n'est fixée ainsi la plateforme correspondante est une plateforme que nous appelons *plateforme virtuelle*.

Au niveau conception, les techniques d'analyse telles que l'analyse d'ordonnançabilité [80] ou analyse de performance [88] requièrent une abstraction des ressources matérielles. En effet, certaines propriétés temporelles telles que le temps d'exécution d'une fonction ou le temps d'accès à une mémoire sont étroitement liées à l'architecture matérielle considérée. Par ailleurs, selon la technique à appliquer, une abstraction de certains aspects de la plateforme logicielle comme la politique d'ordonnancement ou le protocole de synchronisation pour gérer l'accès à une ressource partagée peut s'avérer nécessaire. Afin de garder l'indépendance par rapport à n'importe quel plateforme logicielle d'exécution et rester conforme aux principes du MDA, la plateforme logicielle au niveau conception, est considérée comme une plateforme idéale. Ainsi, la plateforme à ce niveau que nous appelons *plateforme abstraite*, considère des ressources matérielles connues, cependant, aucune limitation des ressources logicielles n'est imposée.

Au niveau implémentation, nous considérons la plateforme d'exécution (matérielle et logicielle) connue. Cette plateforme, que nous appelons *plateforme concrète*, permet d'effectuer des analyses plus précises. La plateforme logicielle à ce niveau est une plateforme logicielle d'exécution sur laquelle l'application est déployée et qui correspond à un système d'exploitation temps réel (RTOS) dans notre cas.

Comme nous l'avons mentionné dans le chapitre précédent, dans cette étude, nous nous intéressons uniquement à la partie logicielle des plateformes. Nous admettons aussi que la

vérification des contraintes de temps au niveau conception fait appel à des techniques d'analyse d'ordonnançabilité. **Par conséquent, dans ce qui suit, les modèles des plateformes abstraites et concrètes décrivent uniquement la partie logicielle des plateformes. En particulier, le modèle de la plateforme concrète correspond à un modèle d'un RTOS décrit à des fins d'analyse d'ordonnançabilité. De plus, nous utilisons le terme *plateforme abstraite d'analyse* pour référencer la plateforme utilisée pour l'analyse d'ordonnançabilité au niveau conception (i.e. cette plateforme offre toutes les ressources logicielles nécessaires pour effectuer une analyse d'ordonnançabilité).**

3.2.2 Identification des besoins pour une plateforme abstraite d'analyse

Selon Almeida [13], une plateforme qui ne sert pas pour l'exécution d'une application, est considérée comme abstraite. De ce fait, une plateforme pour la vérification temporelle au niveau conception, est une plateforme abstraite au sens d'Almeida. En effet, cette plateforme abstraite doit répondre à un certain nombre d'exigences (ou besoins) afin d'accomplir le rôle pour lequel elle a été définie. Ces besoins découlent de l'objectif pour lequel cette plateforme a été définie qui est le déploiement multiplateforme d'une application temps réel critique dans un contexte IDM. Nous identifions ainsi trois exigences :

- ***Req 1*** : la plateforme abstraite d'analyse doit définir tous les concepts nécessaires pour permettre la vérification des propriétés temporelles en utilisant les techniques d'analyse d'ordonnançabilité
- ***Req 2*** : la plateforme abstraite d'analyse doit assurer l'indépendance par rapport au RTOS afin de rester conforme aux principes du MDA
- ***Req 3*** : la plateforme abstraite d'analyse doit partager les mêmes concepts qu'un RTOS afin de préparer la phase de déploiement

Afin de satisfaire ces exigences, la plateforme abstraite d'analyse doit couvrir le maximum de RTOS par la définition et la caractérisation de tous les concepts possibles et qui sont nécessaires pour l'analyse d'ordonnançabilité. A des fins d'analyse, cette plateforme doit jouer le rôle d'un RTOS idéal (sans introduire des contraintes liées à l'implémentation) afin de ne pas limiter les choix du concepteur lors de la description de son application. Par exemple, cette plateforme doit permettre au concepteur de choisir un ordre croissant ou décroissant des niveaux de priorité. Pour ce faire, cette plateforme doit être configurable par le concepteur. Chaque configuration correspond à une plateforme abstraite d'analyse possible. Le test d'analyse à appliquer peut varier selon la configuration (ou selon la plateforme abstraite d'analyse considérée). Par exemple, si le concepteur choisit de configurer sa plateforme abstraite de façon à ce que le protocole de synchronisation PCP [43] est utilisé pour gérer l'accès aux ressources partagées, la formule 2.3 présentée dans le chapitre précédent doit être utilisée pour calculer le temps de blocage d'une tâche, sinon d'autres formules doivent être invoquées [48]. Cependant, lors de la configuration de cette plateforme le concepteur peut tomber sur des situations non significatives. Par exemple, pour l'implémentation d'une ressource partagée, le concepteur choisit un protocole de synchronisation

de type PIP [81] avec une politique de type FIFO pour la file d'attente associée. Ainsi, afin d'éviter de telles situations, des contraintes d'utilisation de la plateforme abstraite d'analyse doivent être considérées.

La mise en œuvre de ces besoins sera détaillée dans la suite de ce rapport au travers la définition d'un langage permettant la description des plateformes logicielles (abstraite et concrète) à des fins d'analyse.

Synthèse

Il découle de ces réflexions que le déploiement d'une application temps réel critique sur un système d'exploitation temps réel (RTOS) dans un contexte IDM se traduit par le raffinement du modèle de conception en un modèle d'implémentation. Le modèle de conception décrit une réalisation possible de l'application en se basant sur une plateforme abstraite d'analyse permettant ainsi la vérification des contraintes temps réel de l'application. Tandis que le modèle d'implémentation correspond à une traduction de cette réalisation pour être spécifique à un RTOS. Plus précisément, ce raffinement revient à transformer les instances applicatives spécifiques aux ressources abstraites sources de la plateforme abstraite d'analyse en des instances applicatives spécifiques aux ressources du RTOS.

3.3 Processus DRIM

En nous basant sur les notions introduites dans la partie précédente, en particulier les modèles de plateformes logicielles à des fins d'analyse : plateforme abstraite d'analyse et plateforme concrète (RTOS), nous introduisons dans cette partie le processus DRIM (Design Refinement toward Implementation Methodology). Ce processus définit un ensemble d'étapes intermédiaires entre la phase de conception et la phase d'implémentation du flot IDM, dans le but d'assurer le déploiement multiplateforme des applications temps réel critiques. Nous commençons tout d'abord par donner un aperçu global sur le processus DRIM et ses différentes étapes. Ensuite, nous expliquons en détails la phase de refactoring qui constitue une étape clé de ce processus.

3.3.1 Vue d'ensemble du processus DRIM

Comme illustré dans la Figure 3.2, le processus DRIM est proposé pour guider le raffinement du modèle de conception d'une application temps réel en un modèle d'implémentation de celle-ci. En effet, comme nous l'avons expliqué dans les chapitres précédents, l'utilisation des techniques d'analyse d'ordonnançabilité pour la vérification des contraintes de temps du modèle de conception nécessite une prise en compte de certains aspects de la plateforme logicielle. Cette prise en compte se traduit par des hypothèses explicitées dans le modèle de la plateforme abstraite d'analyse. Cependant, dans un contexte multiplateforme, ces hypothèses ne sont pas obligatoirement vérifiées pour toutes les plateformes d'exécution cible

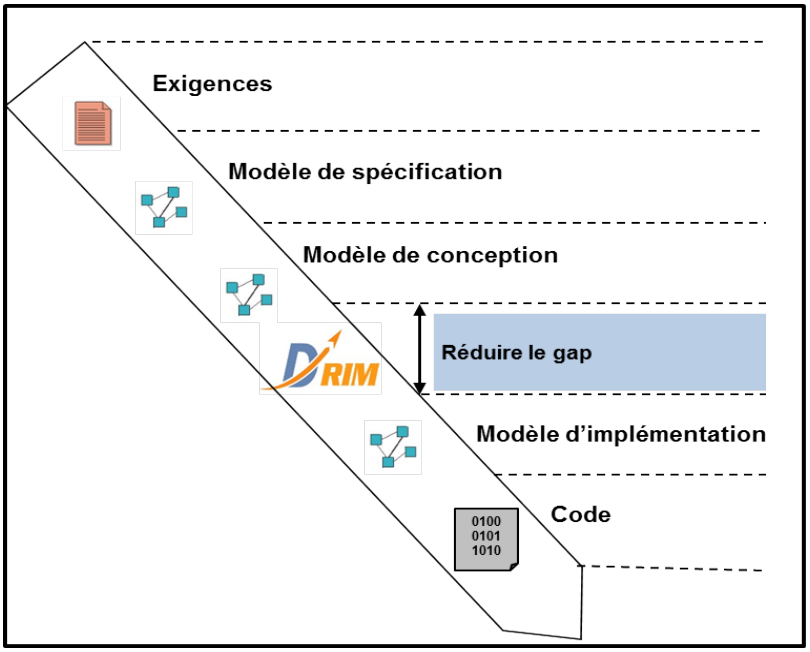


FIGURE 3.2 – *Le processus DRIM dans un flot orienté modèle*

(RTOS) sur lesquelles l'application sera déployée. Ceci peut mener à des modèles de conception qui ne sont pas implémentables sur le RTOS considéré. Par ailleurs, les ressources de la plateforme abstraite d'analyse (sur laquelle se base le modèle de conception) et les ressources du RTOS (sur lequel se base le modèle d'implémentation) peuvent avoir des sémantiques différentes. Il en résulte que le modèle de conception et le modèle d'implémentation n'auront pas la même interprétation ce qui peut affecter les contraintes de temps déjà vérifiées durant la phase de conception et augmenter la distance entre les deux modèles d'une même application. Ainsi, l'objectif du processus DRIM est de réduire cette distance en apportant des solutions à ses différents problèmes dans le but de guider le déploiement d'une application temps réel sur différents RTOS tout en assurant le respect des contraintes de temps. Une vue plus détaillée de ce processus est donnée dans la Figure 3.3.

Le point de départ de ce processus est le modèle de conception initial satisfaisant les contraintes de temps de l'application, le modèle de la plateforme abstraite d'analyse (i.e. un modèle qui décrit une configuration possible de la plateforme abstraite sur laquelle se base les analyses) et le modèle du RTOS cible. La première phase de ce processus est la phase d'évaluation de faisabilité. L'objectif de cette phase est d'évaluer la faisabilité de déploiement du modèle de conception de l'application temps réel analysé en se basant sur la plateforme abstraite (représentée par son modèle), sur le RTOS considéré (représenté aussi par son modèle). Pour ce faire, cette phase se base sur un ensemble de tests prédéfinis qui génèrent des erreurs au cas où des problèmes de déploiement existent.

Dans le cas où, aucun problème de déploiement n'est détecté (i.e. le modèle de conception initial est implémentable), le processus DRIM procède à une phase de portage (*mapping*

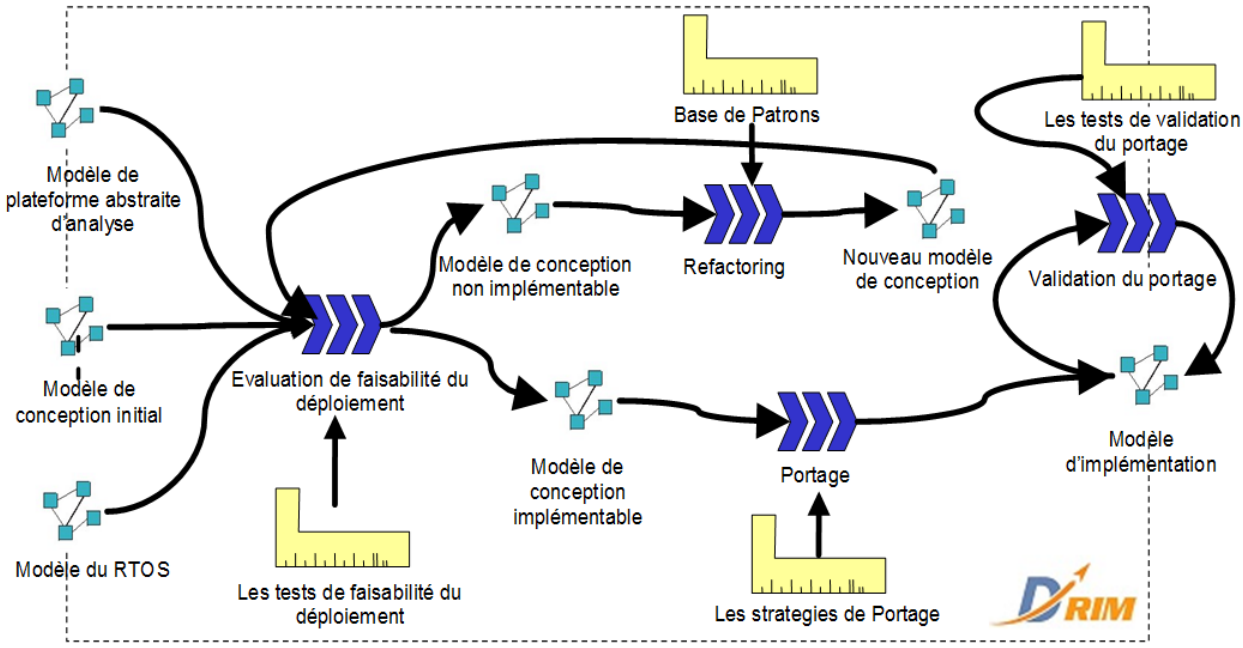


FIGURE 3.3 – *Le processus DRIM*

en anglais) permettant de générer le modèle de l'application spécifique au RTOS cible (modèle d'implémentation). Cette phase de portage définit un ensemble de stratégies, ce qui offre au concepteur la possibilité de générer plusieurs modèles spécifiques à un même RTOS à partir du modèle de conception de l'application. La phase de validation de portage définit un ensemble de propriétés à vérifier sur le modèle d'implémentation résultant. Cette validation a pour objectif de confirmer que le modèle obtenu est conforme au modèle de départ et par la suite que les propriétés de temps ne seront pas affectées.

Dans le cas où la phase d'évaluation de faisabilité détecte un problème de déploiement (i.e. le modèle de conception initial n'est pas implémentable), l'objectif de la phase de refactoring est de trouver une solution implémentable pour le modèle de conception initial. Ceci revient à appliquer le patron approprié à partir d'une base de patrons prédéfinis.

3.3.2 La phase de refactoring

Le processus DRIM passe à l'étape de refactoring si un problème de déploiement du modèle de conception initial est détecté. Ainsi, partant de ce modèle initial satisfaisant les contraintes de temps de l'application sur une plateforme abstraite d'analyse, la phase de refactoring génère un nouveau modèle de conception (voir Figure 3.4) dans le but de résoudre le problème de déploiement signalé. Le modèle de conception généré par cette phase doit garantir deux points :

– **La portabilité** c'est-à-dire le nouveau modèle de conception doit être indépendant du RTOS cible. En d'autres termes, il est au même niveau d'abstraction que le modèle

initial et doit être basé toujours sur la plateforme abstraite d'analyse

– **Le respect des contraintes de temps de l'application temps réel** c'est-à-dire comme le modèle initial, le modèle de conception issue de cette phase doit également satisfaire les contraintes de temps de l'application

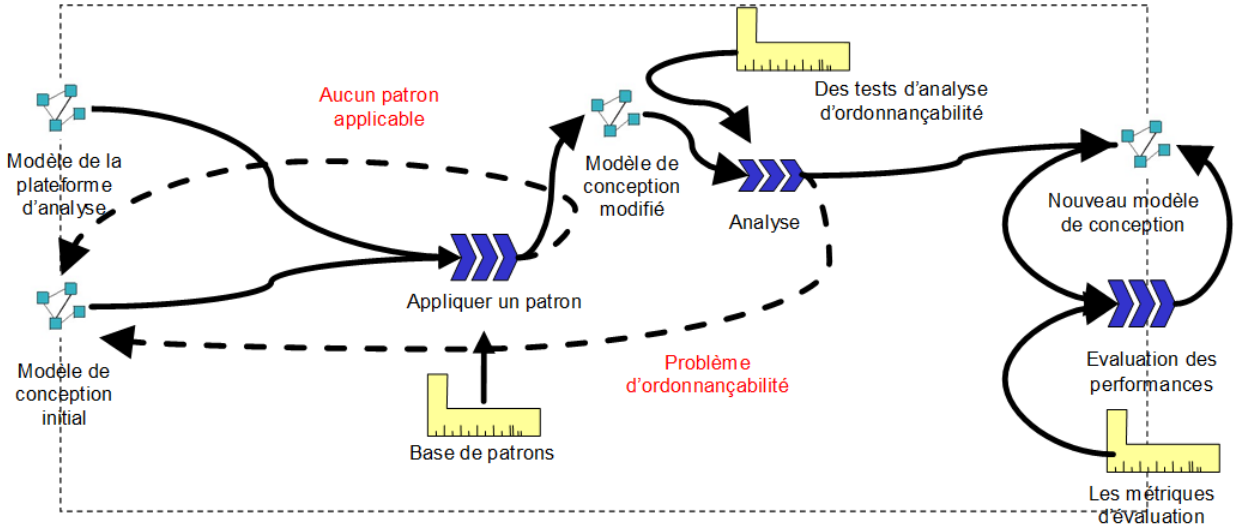


FIGURE 3.4 – *La phase de refactoring*

Comme illustré dans la Figure 3.4, cette phase est basée sur un ensemble de patrons prédéfinis groupés dans une base de patrons. *Nous appelons patron toute transformation de modèles visant à résoudre un problème de déploiement particulier dans certaines hypothèses considérées sur le modèle de conception initial.* Ainsi, lorsqu'un problème de déploiement est détecté, la phase de refactoring commence par chercher dans la base des patrons s'il existe un patron applicable. Un patron est dit applicable si et seulement si :

– Il permet de résoudre le problème de déploiement signalé
– Le modèle de conception initial satisfait les hypothèses définies par ce patron

Dans le cas où la phase de refactoring ne trouve aucun patron applicable dans la base des patrons, une erreur est générée pour informer le concepteur qu'aucune solution n'est trouvée et que le modèle de conception initial est non implémentable sur le RTOS cible.

Sinon, si un patron applicable est trouvé, la phase de refactoring procède à l'application de ce patron et la génération d'un modèle de conception modifié (voir Figure 3.4). Afin de garantir le deuxième point, qui est le respect des contraintes de temps de l'application temps réel, la phase de refactoring intègre une étape d'analyse du modèle de conception modifié. Si les contraintes de temps de l'application ne sont plus respectées pour ce modèle, une erreur est générée pour informer le concepteur que l'application du patron produit des problèmes d'ordonnançabilité et par la suite que le modèle de conception initial n'est pas implémentable sur le RTOS considéré. Dans le cas contraire, c'est-à-dire si les contraintes de temps sont respectées, la phase de refactoring génère le nouveau modèle de conception.

Par ailleurs, une étape d'évaluation des performances est aussi considérée dans la phase de refactoring. Cette étape permet d'estimer les performances du modèle résultant (nouveau modèle de conception) en se basant sur un ensemble de métriques d'évaluation. Comme l'objectif principal de la phase de refactoring est de résoudre des problèmes de déploiement, une dégradation des performances après l'application des patrons peut apparaitre. Par conséquent, cette étape d'évaluation permet d'estimer le coût (en termes de dégradation des performances) du déploiement d'un modèle de conception initial sur un RTOS ou sur une famille d'RTOS donnée [63].

Synthèse

Le processus DRIM montre un ensemble d'étapes partiellement ordonnées et interactifs, qui à partir d'un modèle de conception de l'application et les modèles de plateformes logicielles décrites à des fins d'analyse, produit un modèle d'implémentation. Ce modèle généré représente le modèle de l'application temps réel déployée sur un RTOS et doit satisfaire les contraintes de temps vérifiées au niveau conception.

3.4 Conclusion

Dans ce chapitre, nous avons montré qu'un point clé pour contribuer au déploiement multiplateforme des applications temps réel est la prise en compte explicite des plateformes logicielles. Plus précisément, nous avons proposé de décrire explicitement la plateforme logicielle utilisée au niveau conception, dite abstraite, pour vérifier les contraintes de temps en utilisant les techniques d'analyse d'ordonnançabilité. Contrairement à une plateforme d'exécution logicielle, une plateforme abstraite d'analyse doit permettre la description d'une application au niveau conception à fins d'analyse et préparer son déploiement sur une plateforme d'exécution (RTOS). Ainsi, en nous basant sur cette première contribution, nous avons proposé dans une deuxième partie de ce chapitre le processus DRIM (Design Refinement toward Implementation Methodology). Ce processus introduit un ensemble d'étapes intermédiaires entre la phase de conception et celle d'implémentation du flot IDM dans le but de guider la génération d'un modèle de l'application spécifique à un RTOS à partir du modèle de l'application réalisé au niveau conception satisfaisant les contraintes de temps de celle-ci. Cette génération est basée sur les modèles de plateformes (plateforme abstraite d'analyse et le RTOS) et doit assurer le respect des contraintes de temps en tenant compte des caractéristiques du RTOS cible considéré.

Dans un cadre IDM, la définition d'un processus nécessite la mise en place d'un langage permettant la description des différents modèles intervenants dans ce processus afin de préparer sa mise en œuvre. Ainsi, dans le chapitre suivant, nous introduisons le profil DRIM (une extension du profil MARTE) qui définit les concepts nécessaires pour la description des modèles intervenants dans le processus DRIM et nous présentons une méthodologie pour la description de ces différents modèles.

Chapitre 4

Méthodologie de modélisation dans DRIM

Dans ce chapitre, nous introduisons un langage pour la description des différents modèles intervenant dans le processus DRIM et nous présentons une méthodologie pour la description de ces différents modèles en nous basant sur le langage proposé.

4.1 Introduction

Dans le chapitre précédent nous avons introduit le processus DRIM, qui a pour objectif de guider à partir de la phase de conception le déploiement d'une application temps réel sur différents RTOS en suivant la ligne de l'IDM. L'introduction d'un processus basé modèle, nécessite la définition d'une méthodologie de modélisation basée sur un langage bien structuré permettant ainsi la description des différents modèles du processus et préparant sa mise en œuvre.

Pour cela, dans la première partie de ce chapitre, nous décrivons le profil DRIM (extension du profil MARTE [64]) qui servira comme support pour la description des modèles du processus DRIM. Ensuite, dans une deuxième partie, nous expliquons les règles de modélisation en nous basant sur le profil DRIM proposé.

4.2 Description du langage de modélisation

Cette partie s'intéresse à la définition d'un langage pour la description des modèles du processus DRIM. Nous commerçons tout d'abord par introduire le méta-modèle DRIM qui caractérise les concepts nécessaires pour la description des plateformes logicielles à savoir la plateforme abstraite d'analyse et le RTOS. Ensuite, nous présentons un profil UML que nous appelons DRIM (une extension du profil MARTE) qui définit tous les artéfacts nécessaires pour la description des modèles du processus DRIM.

4.2.1 Le méta-modèle DRIM : un méta-modèle pour la description des plateformes logicielles

Nous définissons dans cette partie le méta-modèle DRIM qui servira comme support pour décrire explicitement les plateformes d'exécution logicielles (plateforme abstraites et RTOS) sur lesquelles repose le processus DRIM. Par la suite, par soucis de simplification, on parlera de plateforme logicielle. Ce méta-modèle doit permettre la satisfaction des besoins identifiés dans 3.2.2 pour la définition d'une plateforme abstraite d'analyse.

Ainsi, afin de satisfaire la première exigence (*Req 1*), le méta-modèle de plateforme proposé caractérise les différentes ressources nécessaires pour l'analyse d'ordonnançabilité d'une application temps réel au niveau conception. En effet, la définition de ces ressources a été basée sur deux études : la première étude concerne les techniques d'analyse d'ordonnançabilité [80], afin d'identifier les besoins de ces techniques en termes d'informations de plateforme. Tandis que l'objectif de la deuxième étude, est de faire une synthèse des caractéristiques de plusieurs RTOS [90] afin de répondre à la deuxième et troisième exigence (*Req 2* et *Req 3*). Dans cette deuxième étude, nous nous sommes surtout focalisés sur les concepts du RTOS nécessaires pour l'analyse d'ordonnançabilité.

Le méta-modèle proposé est présenté dans la Figure 4.1. Ce méta-modèle caractérise une plateforme logicielle *SwPlatform* par son type et son schéma de priorité. Pour le type, re-

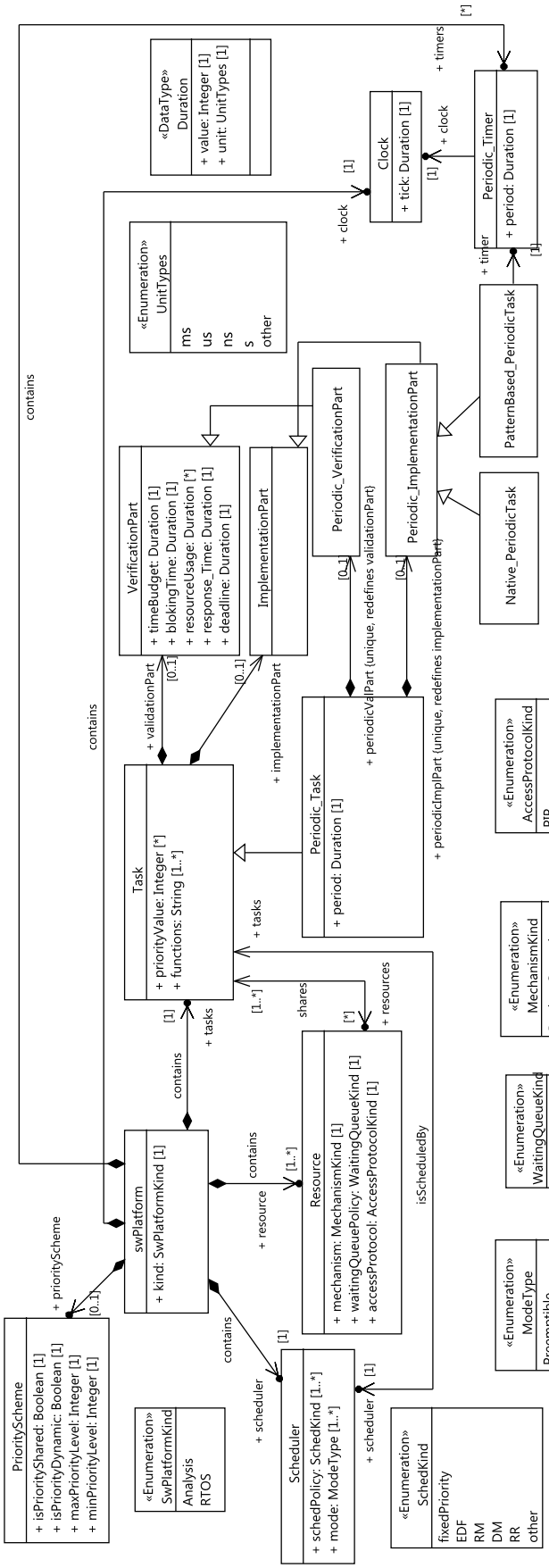


FIGURE 4.1 – *Le méta-modèle DRIM pour la description des plateformes logicielles*

présenté par la propriété *kind* de la classe *SwPlatform*, permet de différencier entre une plateforme d'analyse et une plateforme d'exécution (RTOS). Tandis que le schéma de priorité *PriorityScheme* servira pour la caractérisation de la notion de priorité dans une plateforme logicielle. En effet, un exécutif temps réel dont l'ordonnanceur est basé sur la notion de priorité, définit en général un niveau maximal et un niveau minimal de priorité représentés respectivement par les propriétés *maxPriorityLevel* et *minPriorityLevel* de la classe *PriorityScheme*. Ces valeurs varient selon le RTOS, ainsi l'ordre de priorité peut être croissant ou décroissant. Dans RTEMS [8], par exemple, la valeur 1 représente le niveau de priorité le plus important par contre dans FreeRTOS [1] la valeur 1 représente le niveau le plus faible. De plus, un exécutif temps réel peut autoriser ou pas le partage d'un même niveau de priorité entre les tâches. La propriété *isPriorityShared* de la classe *PriorityScheme* permet de capturer cette information. Donc, si cette propriété est égale à *false*, le RTOS correspondant ne supporte pas le partage des niveaux de priorité entre les tâches. MicroC-OS/II [10] est un exemple de cette famille de RTOS. Enfin, la propriété *isPriorityDynamic* de la classe *PriorityScheme* permet de définir si un RTOS supporte ou pas la variation de la valeur de priorité lors de l'exécution. Pour la plateforme abstraite d'analyse, comme étant un RTOS idéal (de point de vue analyse), le concepteur est capable de configurer le schéma de priorité qu'il souhaite selon ses besoins et/ou selon les capacités de l'outil d'analyse à utiliser.

Une plateforme logicielle est composée d'un ordonnanceur qui est défini par sa politique d'ordonnancement et par son mode (*Préemptif* ou *non Préemptif*). Un exécutif temps réel peut offrir une ou plusieurs politiques d'ordonnancement et un ou deux modes. Par exemple, l'exécutif temps réel RTEMS intègre trois politiques d'ordonnancement : ordonnancement basé sur les priorités fixes [52], Rate Monotonic (RM) [54] et partage de temps (RR) et permet l'utilisation des deux modes. Par contre MicroC-OS/II fournit une seule politique basée sur les priorités fixes avec un seul mode préemptif. Une plateforme abstraite d'analyse doit permettre au concepteur de choisir n'importe quelle politique et n'importe quel mode. Cependant, pour une configuration donnée, une plateforme d'analyse doit définir une seule politique d'ordonnancement et un seul mode. Afin d'assurer cette condition, nous définissons sur le méta-modèle DRIM la contrainte OCL [7] présentée dans la Figure 4.2

```
context swPlatform
inv:
(self.kind = #Analysis) implies ((self.scheduler.schedPolicy →
size())=1) and ((self.scheduler.mode →size())=1)
```

FIGURE 4.2 – *Une contrainte OCL sur le méta-modèle DRIM pour limiter le choix de la politique d'ordonnancement et du mode d'un ordonnanceur pour une configuration d'une plateforme abstraite d'analyse*

Pour une plateforme logicielle, une tâche applicative (*Task*) peut servir soit pour l'ana-

lyse (*VerificationPart*), soit pour l'implémentation ou l'exécution (*ImplementationPart*). Dans les deux cas, cette tâche est caractérisée par sa valeur de priorité (pour un ordonnanceur basé sur les priorités), le/les fonctions applicatives qu'elle implémente et elle gérée par un ordonnanceur. Une tâche d'analyse est définie en plus par les propriétés nécessaires pour l'analyse d'ordonnançabilité telles que son échéance *deadline*, son temps d'exécution *timeBudget*, etc. Une tâche périodique *Periodic_Task* est un type particulier de tâche avec une propriété supplémentaire qui est la période *period*. De même une tâche périodique peut servir pour la vérification *Periodic_VerificationPart* ou l'implémentation *Periodic_ImplementationPart*.

Un exécutif temps réel peut fournir le concept de tâche périodique par défaut tel qu'Osek Time [4] *Native_PeriodicTask*. Dans le cas contraire, une implémentation à base de motifs de conception [34] doit être considérée *PatternBased_PeriodicTask* faisant intervenir un Timer périodique et l'horloge fournie par cet exécutif.

Par ailleurs, une plateforme logicielle fourni une ou plusieurs implémentations d'une ressource partagée. Chaque implémentation possible revient à combiner les trois propriétés qui caractérisent cette ressource et qui sont : *mechanism*, *waitingQueuePolicy* et *accessProtocol*. Un exécutif temps réel peut fournir une ou plusieurs implémentations possibles d'une ressource partagée. Tandis qu'une plateforme abstraite d'analyse doit permettre toutes les implémentations possibles afin de ne pas limiter les choix du concepteur. Cependant, pour une configuration donnée d'une plateforme abstraite d'analyse, une seule implémentation doit être considérée (i.e. toutes les ressources sont implémentées en utilisant les mêmes choix dans un modèle de tâches). Ceci se traduit par la contrainte OCL donnée par la Figure 4.3. Selon l'implémentation choisie par le concepteur, la technique d'analyse peut varier, en particulier la technique de calcul du temps de blocage [48].

```
context swPlatform
inv:
(self.kind = #Analysis) implies ((self.resource → size()) =1)
```

FIGURE 4.3 – *Une contrainte OCL sur le méta-modèle DRIM pour définir une seule implémentation des ressources partagées pour une configuration donnée d'une plateforme abstraite d'analyse*

Certaines combinaisons de ces trois propriétés (*mechanism*, *waitingQueuePolicy* et *accessProtocol*) de la classe *Resource* peuvent ne pas correspondre à des situations réelles d'implémentation d'une ressource partagée. Afin d'éviter ces situations, une contrainte OCL sur la classe *Resource* de ce méta-modèle peut être associée à chaque situation non-significative. Un exemple d'une telle situation se traduit par le scénario suivant : un mécanisme d'accès de type sémaphore booléenne, une politique de la file associée est de type FIFO et un protocole d'accès (de synchronisation) de type PCP. Ainsi la contrainte OCL qui permet d'éviter une telle situation est donnée par la Figure 4.4.

La définition du méta-modèle DRIM, présenté dans la Figure 4.1, se limite aux ressources nécessaires pour assurer l'analyse et le déploiement des applications visées dans cette étude.

```
context Resources
inv:
(Self.mechanism = #BooleanSemaphore) and (Self.waitingQueuePolicy =
#FIFO) implies (not (Self.concurrentAccessProtocol = #PCP)
```

FIGURE 4.4 – *Une contrainte OCL sur le méta-modèle DRIM pour éliminer une situation non significative pour l'implémentation des ressources partagées*

D'autres ressources qui sont nécessaires pour l'analyse et le déploiement des applications plus complexes ne sont pas représentées dans ce méta-modèle, comme les mécanismes de communication par exemple.

4.2.2 Le profil DRIM : une extension du profil MARTE

L'objectif de cette partie est de définir un profil UML pour la description des modèles intervenants dans le processus DRIM. Pour cela, nous commençons par proposer une implantation du méta-modèle DRIM dans un profil UML qui constituera un sous-profil du profil DRIM destiné à la modélisation des plateformes logicielles. Ensuite, dans un deuxième sous-profil du profil DRIM, nous introduisons les artéfacts nécessaires pour la création des modèles des applications dans le processus DRIM.

4.2.2.1 Implantation du méta-modèle DRIM dans un profil UML

L'utilisation du standard UML pour la création des modèles de plateformes logicielles se traduit par l'implantation du méta-modèle de plateformes, DRIM, dans un profil UML. En effet, dans le domaine du temps réel et de l'embarqué, le profil UML-MARTE [64] identifie, au travers de son sous-profil Software Resource Modeling (SRM), un ensemble de concepts utiles à la description des plateformes logicielles. Ainsi, ce profile implante la majorité des concepts définit dans le méta-modèle proposé. Néanmoins, SRM a été définit principalement pour modéliser des plateformes d'exécution logicielle. Par conséquent, la modélisation des plateformes logicielles abstraites pour l'analyse (ou pour une autre utilisation) ne peut pas être totalement assurée par SRM. Ceci nous amène, d'une part, à utiliser d'autres concepts de MARTE dont la sémantique coïncide avec certains concepts du méta-modèle proposé. D'autre part, nous définissons de nouveaux artéfacts dans le but d'implanter/préciser les concepts du méta-modèle qui ne possèdent pas de concepts équivalents dans MARTE.

Le tableau 4.1 illustre la relation de correspondance entre les concepts du méta-modèle proposé et les concepts de MARTE.

Un ordonnanceur représenté par la classe *Scheduler* dans le méta-modèle correspond au stéréotype «Scheduler» du sous-profil Generic Resource Modeling (GRM) de MARTE qui offre les ressources génériques pour la modélisation d'une plateforme (matérielle ou logicielle, abstraite ou concrète). D'autre part, la classe *Resource* du métamodèle DRIM re-

TABLE 4.1 – Correspondance entre les concepts du méta-modèle DRIM et les concepts de MARTE

	Concepts du méta-modèle DRIM	Concepts de MARTE
Ressources	Scheduler	GRM : : scheduler
	Resource	SRM : : swMutualExclusionResource
	Periodic_Timer	SRM : : alarm
	Clock	SRM : : swTimerResource
Types	SchedKind	GRM : : SchedPolicyKind
	MechanismKind	SRM : : MutualExclusionResourceKind
	WaitingQueueKind	SRM : : QueuePolicyKind
	AccessProtocolKind	SRM : : ConcurrentAccessProtocolKind
	Duration	BasicNFP_Types : : NFP_Duration
	UnitTypes	TimeLibrary : : TimeUnitKind

présente une ressource logicielle qui peut être partagée entre les tâches. Le stéréotype de SRM dont la sémantique est conforme à ce concept est «SwMutualExclusionResource». Par ailleurs, la sémantique du concept *Clock* du méta-modèle DRIM coïncide avec la sémantique du stéréotype «SwTimerResource» de SRM.

Un point fort du sous-profil SRM est qu'il offre la possibilité de décrire une ressource logicielle avec une sémantique suffisamment précise. Cette sémantique est capturée initialement par le stéréotype et peut être précisée encore par les valeurs des propriétés de ce stéréotype. Ainsi, la sémantique d'une ressource peut varier pour un même stéréotype appliqué selon les valeurs des propriétés. Comme illustré dans la Figure 4.5, la sémantique du concept *Periodic_Timer* du méta-modèle proposé est conforme à la sémantique du stéréotype «Alarm» de SRM avec une valeur par défaut égale à *periodic* pour la propriété *type* de ce stéréotype.

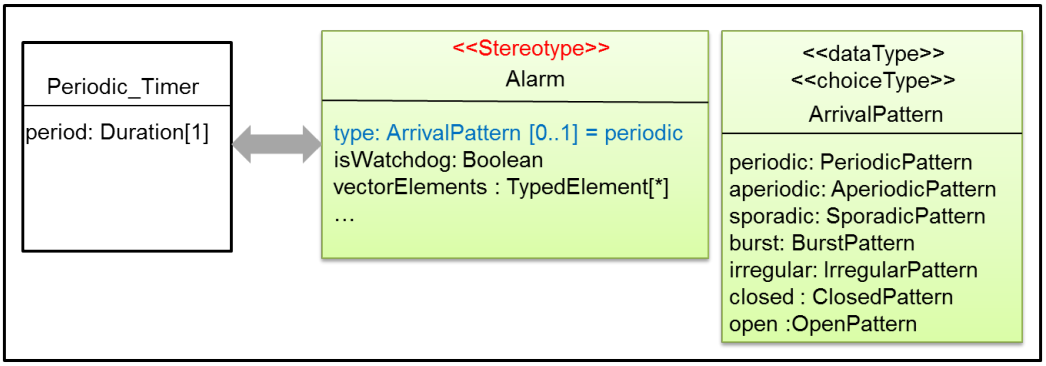


FIGURE 4.5 – *Correspondance entre le concept Periodic_Timer du méta-modèle et le stéréotype «Alarm» de SRM*

Comme le montre le tableau 4.1, tous les types définis dans le méta-modèle proposé ont une implantation équivalente dans MARTE à l'exception des types *ModeType* et *SwPlatform-*

Kind. La définition du mode de l'ordonnanceur dans SRM est représentée par une propriété booléenne *isPreemptible* au niveau du stéréotype «Scheduler» qui permet de différencier les deux modes. Ainsi, cette solution est adoptée et l'implantation de ce type (*ModeType*) s'avère non nécessaire. De plus, la caractérisation d'une plateforme logicielle selon le rôle joué dans un flot dirigée par les modèles n'as pas été clairement traitée dans SRM. Ce qui justifie l'absence, dans la spécification de SRM, du type *SwPlatformKind* et d'un concept dont la sémantique est conforme au concept *SwPlatform* du méta-modèle DRIM. Dans ce même contexte, une tâche dans SRM correspond seulement à une tâche d'exécution. Par conséquent, la sémantique du stéréotype «swSchedulableResource» qui référence une tâche dans SRM n'est pas conforme à la sémantique du concept «Task» (tâche) du méta-modèle DRIM puisque ce dernier peut servir en plus pour la vérification temporelle (*VerificationPart*). En outre, la notion de priorité est représentée de façon très abstraite dans SRM par la propriété *PriorityElements* du stéréotype «SwSchedulableResource». Ainsi, une implantation concrète du concept *PriorityScheme* du méta-modèle n'a pas été définie dans SRM.

Afin de pallier à ce manque, le profil DRIM implante les concepts du méta-modèle DRIM qui n'ont pas une implantation alternative dans MARTE. Ceci consiste à créer de nouveaux stéréotypes ou à spécialiser des stéréotypes qui existent déjà dans MARTE afin d'avoir la sémantique voulue. Ainsi, le concept d'une plateforme logicielle *swPlatform* du méta-modèle DRIM est représenté par un stéréotype qui porte le même nom dans le profil DRIM. Ce stéréotype étend la méta-classe *Paquetage* d'UML et définit une propriété unique appelée aussi *kind*. Cette propriété est typée par le type énuméré *SwPlatformKind* permettant ainsi de caractériser une plateforme logicielle comme étant une plateforme abstraite d'analyse ou un RTOS. A chaque plateforme logicielle, nous pouvons associer un schéma de priorité «priorityScheme» représenté par un stéréotype qui étend la méta-classe *classe* d'UML. Ce stéréotype définit les propriétés du concept *PriorityScheme* du méta-modèle DRIM. Le stéréotype «swTask» spécialise le stéréotype «swSchedulableResource» de SRM dans le but d'avoir la même sémantique du concept *Task* du méta-modèle DRIM. Ainsi, ce stéréotype référence toute entité ordonnançable qu'elle soit utilisée pour la vérification (*VerificationPart*) ou pour l'implémentation (*ImplementationPart*). Ce stéréotype référence aussi toute entité ordonnançable périodique (*Periodic_Task*) qu'elle soit utilisée pour la vérification (*Periodic_VerificationPart*) ou pour l'implémentation («*Periodic_ImplementationPart*»).

La projection du méta-modèle DRIM dans un profil UML, se traduit ainsi par la définition d'un sous-profil du profil DRIM que nous appelons PM (Platform_Modeling). Ce sous-profil, présenté dans la figure 4.6, réutilise quelques concepts de MARTE et introduit de nouveaux concepts dans le but de fournir touts les artéfacts nécessaires pour la modélisation des plateformes logicielles qui représentent des entrées pour le processus DRIM.

4.2.2.2 Identification des concepts pour la modélisation des applications dans DRIM

Dans le processus DRIM, une application est modélisée à deux niveaux d'abstraction différents : Au niveau conception, le modèle de l'application (modèle de conception) décrit

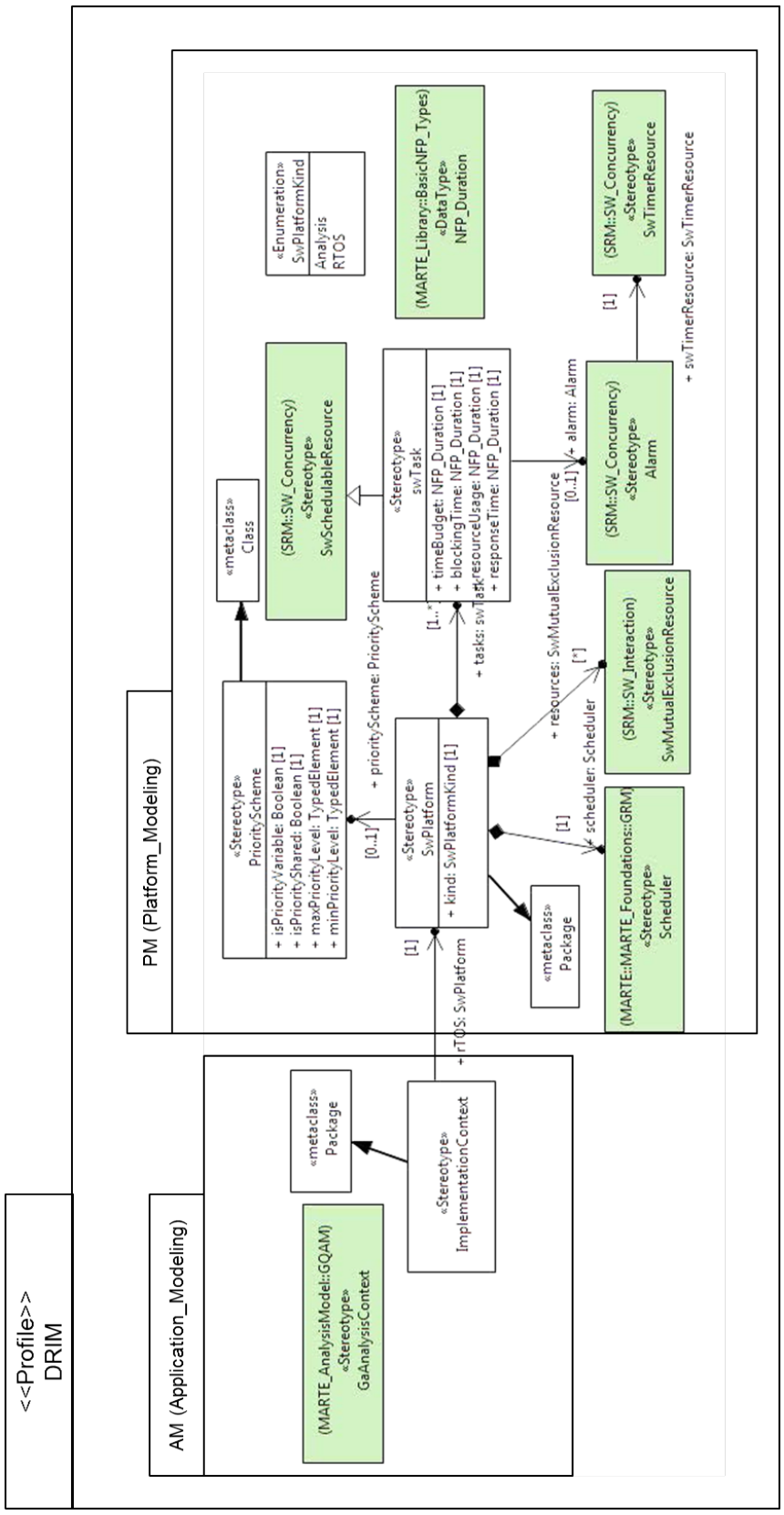


FIGURE 4.6 – *Le profil DRIM : une extension du profile MARTE*

une réalisation possible de l'application en se basant sur une plateforme abstraite d'analyse. Une analyse de cette réalisation sera par la suite effectuée pour vérifier si les contraintes de temps sont bien respectées. Au niveau implémentation, le modèle de l'application (modèle d'implémentation) décrit la réalisation de l'application sur un noyau temps réel (RTOS) en se basant sur le modèle de ce noyau. Ainsi, le modèle de l'application au niveau implémentation est un modèle de celle-ci spécifique à un RTOS. Nous avons ainsi identifiés les différents concepts nécessaires pour la description des applications dans le processus DRIM. Comme le monte la Figure 4.6, ces concepts sont regroupés dans un sous-profil du profile DRIM que nous appelons AM (Application_Modeling).

Le modèle de conception est créé pour décrire une application à des fins d'analyse. Pour cela, nous utilisons le stéréotype «GaAnalyisContext» du sous-profil GQAM (Generic Quantitative Analysis Modeling) de MARTE, qui permet de référencer un contexte d'analyse, pour annoter le modèle de conception d'une application.

En ce qui concerne le modèle d'implémentation, nous définissons un nouveau stéréotype que nous appelons «ImplementationContext» qui étend la méta-classe *Paquetage* d'UML. Ce stéréotype sert pour annoter/tagger les modèles d'application au niveau implémentation et permet de référencer la plateforme concrète (RTOS) pour chaque contexte d'implémentation(i.e. association entre les stéréotypes «ImplementationContext» et «swPlataform»).

Synthèse

Le profil DRIM que nous avons introduit dans cette partie servira de support pour la description des modèles du processus DRIM. Cependant, il est à noter que la Figure 4.6 **ne représente qu'un extrait de ce profil qui montre uniquement les concepts nécessaires pour le déploiement des applications visées dans cette étude.** Ainsi, pour des applications plus complexes telles que par exemple les applications basées sur des architectures distribuées faisant intervenir des mécanismes de communications, d'autres concepts doivent être considérés. Dans ce qui suit, nous nous basons sur ce profil pour expliquer les différentes règles de description des modèles du processus DRIM.

4.3 Règles méthodologiques de modélisation

Cette partie définit les règles de description des modèles du processus DRIM en utilisant le profil DRIM. Nous commençons par la définition des règles de modélisation des plateformes logicielles. Ensuite, nous expliquons les règles de modélisation des applications temps réel.

4.3.1 Les règles de modélisation de plateformes

La définition des règles de modélisation passe par l'identification, d'une part, des éléments UML mis en œuvre pour la création des modèles de plateformes visées dans cette

étude. D'autre part, par la définition des heuristiques d'utilisation des stéréotypes (du profil DRIM) afin de donner une sémantique aux différents éléments UML identifiés. Dans ce travail, nous nous intéressons à la description structurelle des plateformes logicielles. Ainsi, afin de modéliser une plateforme logicielle, nous nous basons sur les diagrammes de classe UML permettant de décrire la structure de celle-ci. Nous détaillerons dans ce qui suit les règles à suivre pour créer les modèles de plateformes logicielles visées dans cette étude.

4.3.1.1 Les règles liées aux éléments UML

Les règles définies dans ce paragraphe expliquent l'utilisation des éléments UML (en particulier les éléments du diagramme de classe) pour créer les modèles de plateformes logicielles.

– **Règle P1 :** *Une plateforme logicielle est représentée par un Paquetage UML qui porte le nom de la plateforme considérée.*

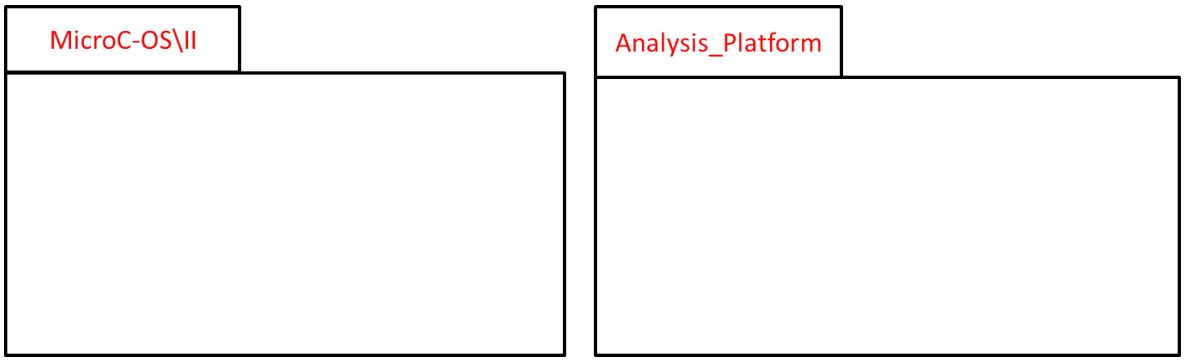


FIGURE 4.7 – *Exemple d'application de la première règle pour MicroC-OS/II et une plateforme d'analyse*

La Figure 4.7 montre l'application de cette règle pour le RTOS MicroC-OS/II [10] et pour une plateforme abstraite d'analyse. En effet, le concepteur est libre de choisir le nom de la plateforme d'analyse qu'il souhaite. Ce nom peut être associé aussi à l'outil d'analyse à utiliser pour vérifier les contraintes de temps.

– **Règle P2 :** *Chaque ressource qui caractérise la plateforme logicielle est représentée par une Classe UML portant un nom significatif pour refléter la sémantique de la ressource associée.*

La Figure 4.8 illustre l'application de cette règle pour MicroC-OS/II et une plateforme d'analyse. En effet, pour modéliser les deux ressources ordonnanceur et entité ordonnançable (tâche) d'une plateforme logicielle, deux classes sont créés dans chaque paquetage. Une tâche d'exécution MicroC-OS/II (respectivement une tâche abstraite de la plateforme d'analyse) est modélisée par une classe appelée *MicroC_Task* (respectivement une classe appelée *Analysis_Task*). De la même façon, un ordonnanceur MicroC-OS/II (respectivement un ordonnanceur de la plateforme d'analyse) est modélisé par une classe appelée

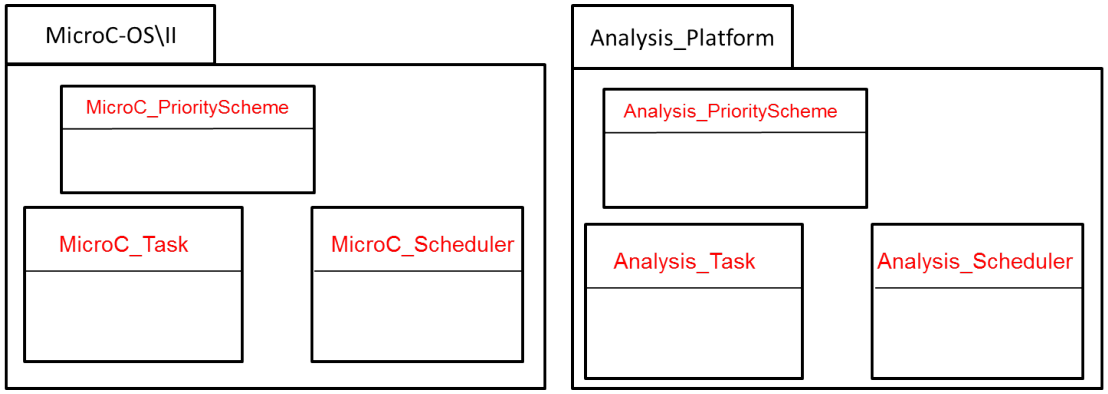


FIGURE 4.8 – *Exemple d'application de la deuxième règle pour MicroC-OS/II et une plateforme d'analyse*

MicroC_Scheduler (respectivement une classe appelée *Analysis_Scheduler*). Par ailleurs, deux classes appelées *MicroC_PriorityScheme* et *Analysis_PriorityScheme* sont créés pour référencer respectivement le schéma de priorité associé au RTOS MicroC-OS/II et le schéma de priorité associé à la plateforme abstraite d'analyse.

– **Règle P3 :** *Chaque classe du modèle de la plateforme logicielle définit l'ensemble des propriétés nécessaires pour caractériser le concept associé. Les propriétés à définir pour chaque classe sont identifiées dans le méta-modèle de plateforme proposé.*

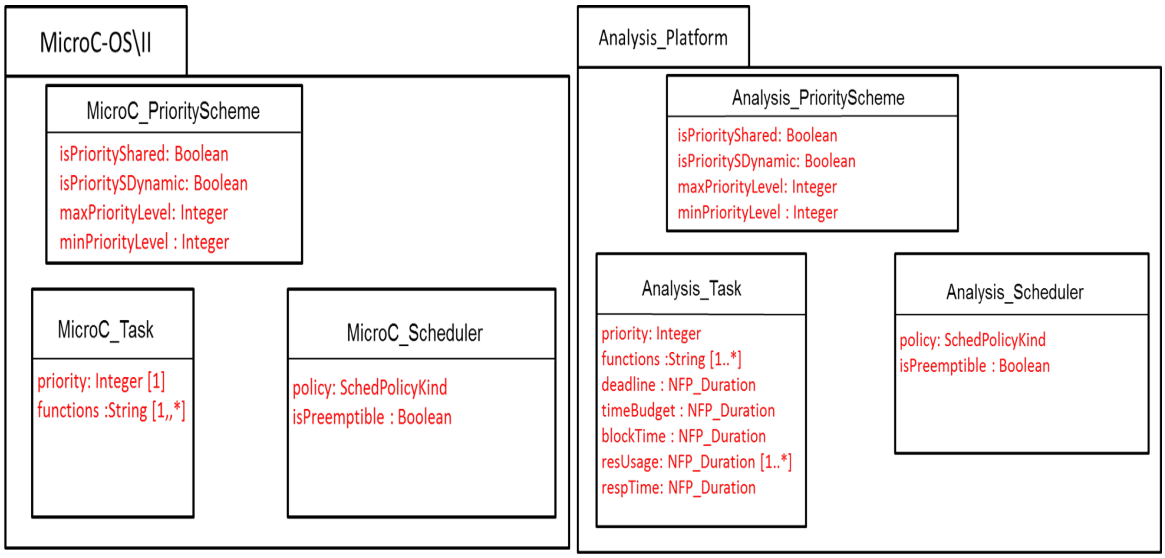


FIGURE 4.9 – *Exemple d'application de la troisième règle pour MicroC-OS/II et une plateforme d'analyse*

Par exemple, la sémantique de la classe *MicroC_Task* est conforme à la sémantique du concept *ImplementationPart* du méta-modèle DRIM. Ce qui justifie le fait que cette classe définit juste les deux propriétés *priority* et *functions*. Tandis que la sémantique de la classe *Analysis_Task* est conforme à la sémantique du concept *VerificationPart* du méta-modèle DRIM. De ce fait, la classe *Analysis_Task* définit les mêmes propriétés que le concept *VerificationPart* du méta-modèle.

– **Règle P4 :** *Les relations entre les ressources d'une plateforme logicielle se traduisent par des associations entre les classes correspondantes.*

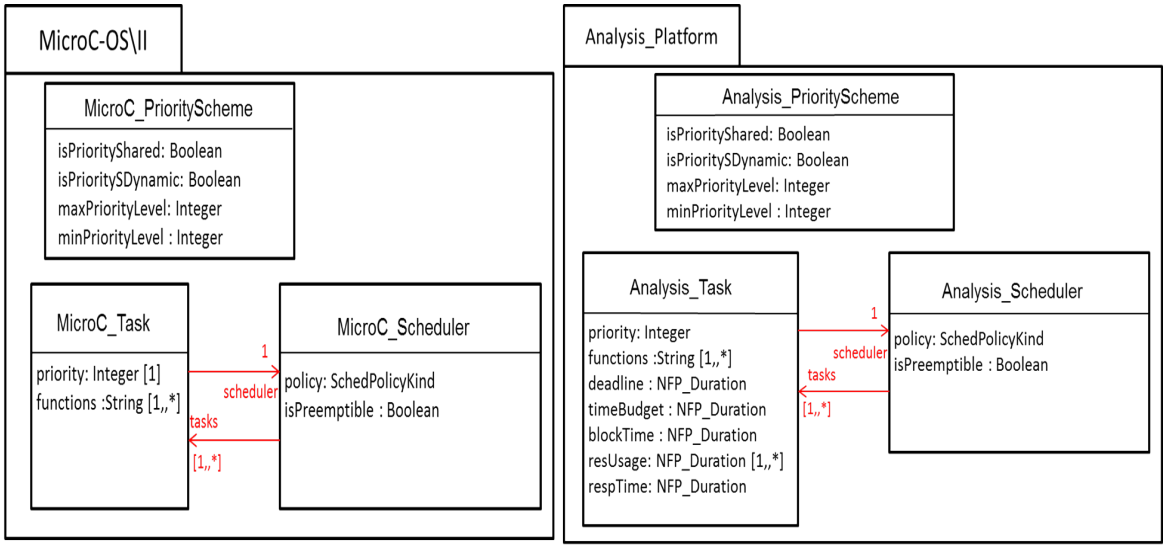


FIGURE 4.10 – *Exemple d'application de la quatrième règle pour MicroC-OS/II et une plateforme d'analyse*

Dans MicroC-OS/II, la planification de l'exécution des différentes tâches doit être gérée par un ordonnanceur. Cette relation se traduit par les deux associations entre les deux classes *MicroC_Task* et *MicroC_Scheduler* comme le montre la Figure 4.10. De même, la vérification des propriétés temporelles des tâches abstraites décrivant l'application suppose que ces tâches sont gérées par un ordonnanceur. Ce qui justifie aussi les associations entre les classes *Analysis_Task* et *Analysis_Scheduler*.

Pour un noyau temps réel, les valeurs par défaut des propriétés au niveau de son modèle caractérisent les ressources offertes par ce noyau (en d'autres termes traduisent les capacités de ce noyau). Tandis que les valeurs par défaut des propriétés d'une plateforme d'analyse correspondent à une configuration possible de la plateforme abstraite (en d'autres termes elles traduisent les choix du concepteur) dans le but de permettre la vérification des propriétés temporelles de l'application. En effet, les valeurs par défaut des propriétés qui caractérisent une plateforme d'analyse peuvent être modifiées par le

concepteur afin de configurer une autre plateforme et appliquer par la suite d'autres types d'analyse. Par contre, les valeurs par défaut du modèle d'un RTOS sont fixes car ils traduisent la spécification de ce dernier. Ceci nous amène à définir la cinquième règle suivante :

– **Règle P5 :** *Les propriétés des classes qui caractérisent la plateforme logicielle peuvent avoir une valeur par défaut dans son modèle.*

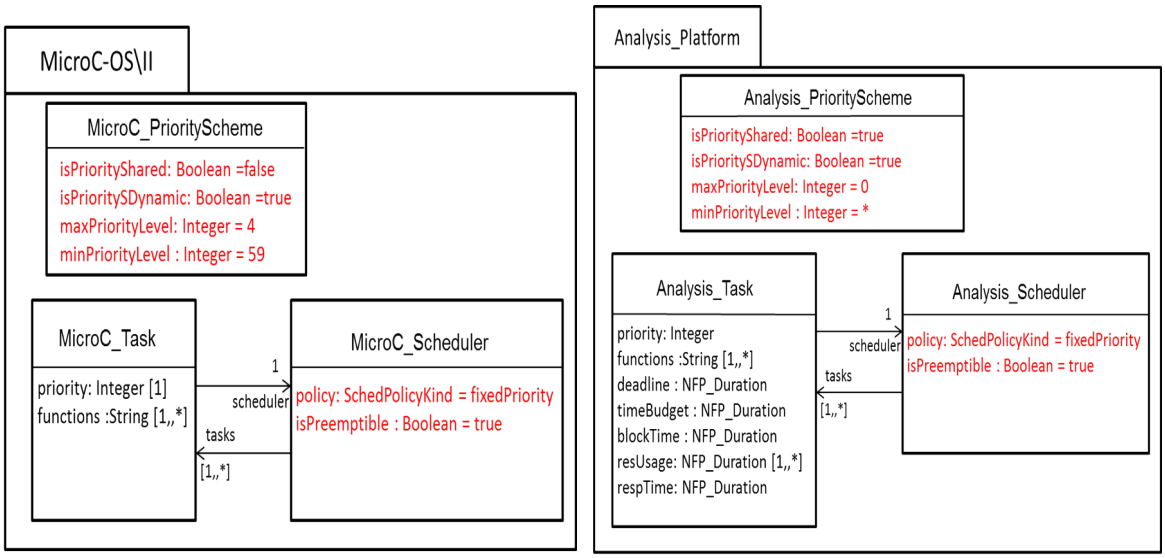


FIGURE 4.11 – *Exemple d'application de la cinquième règle pour MicroC-OS/II et une plateforme d'analyse*

L'identification de cette règle permet de mettre en évidence la différence entre les deux types de plateforme (plateforme d'analyse et plateforme d'exécution). Un exemple d'application de cette règle est donné dans la Figure 4.11. En effet, le noyau MicroC-OS/II est caractérisé par un ordonnanceur préemptif avec une politique à base de priorités fixes. Comme le montre la figure 4.11, ceci se traduit au niveau du modèle de la plateforme MicroC-OS/II par des valeurs par défaut des propriétés *policy* et *isPreemptible*. D'autre part, le schéma de priorité d'une plateforme logicielle constitue une particularité de cette dernière. Ainsi, toutes les propriétés du concept *MicroC_PriorityScheme* doivent avoir une valeur par défaut au niveau du modèle de MicroC-OS/II. En ce qui concerne la plateforme d'analyse, un exemple de configuration de cette dernière est donné dans la Figure 4.11. La plateforme considérée dispose d'un ordonnanceur préemptif de type priorités fixes. Par ailleurs, cette plateforme offre un nombre illimité de niveaux de priorité distinct allant de 0 à l'infini. En outre, elle autorise le partage d'un même niveau de priorité entre les tâches et le changement de la valeur de la priorité d'une tâche au cours de l'exécution.

Dans certains cas, la description d'une ressource logicielle d'exécution nécessite la mise en œuvre de plusieurs autres ressources dont la sémantique globale correspond à la

sémantique de cette dernière. En termes de modélisation, ceci se traduit par l'utilisation des motifs de conception [34]. En UML, il existe deux approches principales pour la description structurelle des motifs de conception : une approche basée sur l'utilisation des collaborations et une approche basée sur l'utilisation des classes structurées. Cette description structurelle montre, pour les deux approches, un ensemble d'instances de classes et la relation entre eux dans le but de réaliser un concept particulier. Bien que ces deux approches permettent une description structurelle des motifs de conception, les classes structurées possèdent l'avantage de permettre la description des opérations. Ceci nous amène à la définition d'une sixième règle qui est la suivante :

– **Règle P6 :** *Dans le cas où la description d'une ressource de la plateforme logicielle nécessite la définition d'un motif de conception, une approche basée sur l'utilisation des classes structurées sera adoptée.*

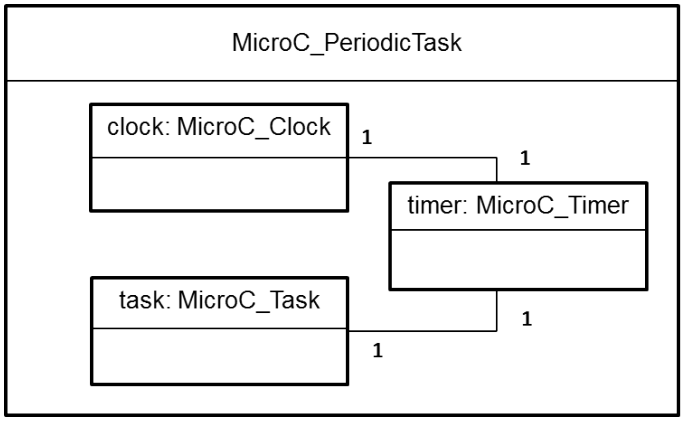


FIGURE 4.12 – *Exemple de motif de conception pour la description d'une tâche périodique dans MicroC-OS/II*

La Figure 4.12 montre un exemple de motif de conception à base des classes structurées pour la description d'une classe périodique dans MicroC-OS/II. En effet, cette tâche périodique correspond à une tâche ordinaire MicroC-OS/II *MicroC_Task* activée périodiquement par un Timer *MicroC_Timer* qui à son tour est géré par une horloge *MicroC_Clock*.

Ainsi, l'application de ces règles constitue la première phase de modélisation des plateformes logicielles. A l'issue de cette phase, une description structurelle de la plateforme est élaborée. Cette description est interprétable seulement par son concepteur ce qui limite son intégration dans une méthodologie générative orientée modèles (i.e. la sémantique des différents concepts qui décrivent la plateforme ne peut pas être détectée par un outil).

4.3.1.2 Les règles de stéréotypage

Cette deuxième phase de modélisation des plateformes consiste à appliquer le profil DRIM dans le but de donner une sémantique, interprétable par un outil, aux différents

modèles issus de la première phase. L'application du profil DRIM revient à identifier et appliquer les stéréotypes appropriés aux différents concepts du modèle de plateforme d'une part et à préciser les valeurs de leurs propriétés d'autre part. Ceci permet la réutilisation des modèles de plateformes et leur intégration dans une méthodologie générative. Nous identifions ainsi les trois règles suivantes :

– **Règle P7 :** *Chaque concept du modèle de plateforme est annoté par le stéréotype du profil DRIM dont la sémantique est conforme à la sémantique du concept considéré.*

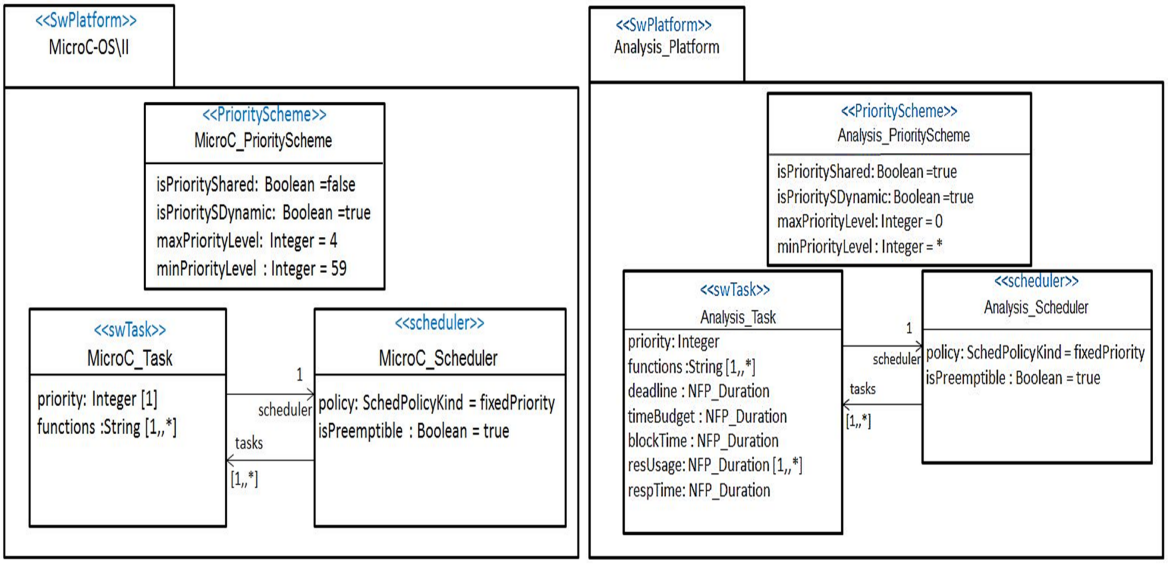


FIGURE 4.13 – *Exemple d'application de la septième règle pour MicroC-OS/II et une plateforme d'analyse*

L'application de cette règle aux modèles des plateformes issus de la phase précédente est donnée dans la Figure 4.13. En effet, le paquetage représentant la plateforme logicielle (plateforme abstraite ou RTOS) est annoté par le stéréotype «swPlatform» du profil DRIM. Par ailleurs, le schéma de priorité de chaque plateforme est annoté par le stéréotype «priorityScheme». La classe représentant l'ordonnanceur est stéréotypée par le stéréotype «scheduler» pour la plateforme d'analyse et MicroC-OS/II. En outre, les entités ordonnançables du noyau MicroC-OS/II et de la plateforme d'analyse sont annotées par le stéréotype «swTask» du profil DRIM.

– **Règle P8 :** *Une propriété du stéréotype référence un attribut de la classe auquel il est appliqué si et seulement si leurs sémantiques coïncident.*

Cette règle se traduit par une correspondance entre les attributs d'une classe et les propriétés du stéréotype appliqué à cette classe. Par exemple, dans la Figure 4.14, la propriété *priorityElements* du stéréotype «swTask» correspond à l'attribut *priority* de la classe *MicroC_Task*. Par ailleurs, la sémantique de la propriété *entryPoints* du stéréotype «swTask»

est conforme à la sémantique de l'attribut *functions* de la classe *MicroC_Task*.

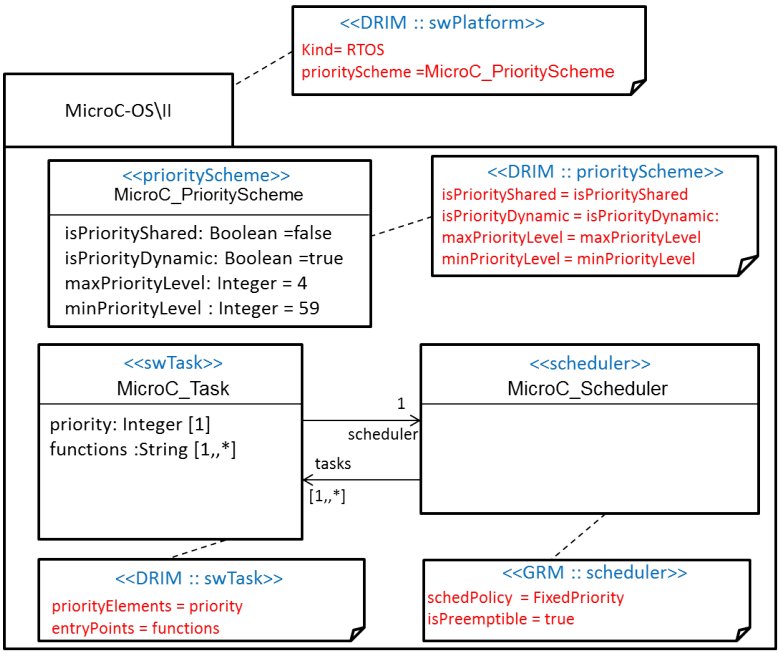


FIGURE 4.14 – *Exemple d'application de la huitième et neuvième règle pour MicroC-OS/II*

De même, dans la Figure 4.15, le stéréotype «swTask» appliqué à une tâche d'analyse référence les attributs *priority* et *functions* de la classe *Analysis_Task* respectivement par les propriétés *priorityElements* et *entryPoints*. Les autres attributs de la classe *Analysis_Task* et qui serviront pour l'analyse sont référencés également par les propriétés correspondantes du stéréotype «swTask».

Du fait de l'implémentation de MARTE (le profil DRIM est une extension de MARTE), certaines propriétés des stéréotypes ne peuvent pas référencer des attributs de la classe au quelle ils sont appliqués (par exemple les propriétés typés par un énuméré ou de type boolean). Dans ce cas, les valeurs de ces attributs sont affectées au niveau profile. Citons comme exemples, les deux propriétés *schedPolicy* et *isPreemptible* du stéréotype «scheduler». Comme illustré dans les Figures 4.14 et 4.15, les valeurs par défaut de ces deux propriétés sont données au niveau profil.

– **Règle P9 :** *L'application du stéréotype «swPlatform» au paquetage représentant la plateforme permet de préciser le type (RTOS ou plateforme d'analyse) et le schéma de priorité associer à cette dernière.*

Ainsi, dans le modèle de MicroC-OS/II (respectivement le modèle de la plateforme d'analyse) donné dans la Figure 4.14 (respectivement dans la Figure 4.15), le type de la plateforme est spécifié dans la propriété *kind* du stéréotype «swPlatform» et qui est égale à RTOS (respectivement à Analysis). D'autre part, le schéma de priorité est capturé dans la propriété

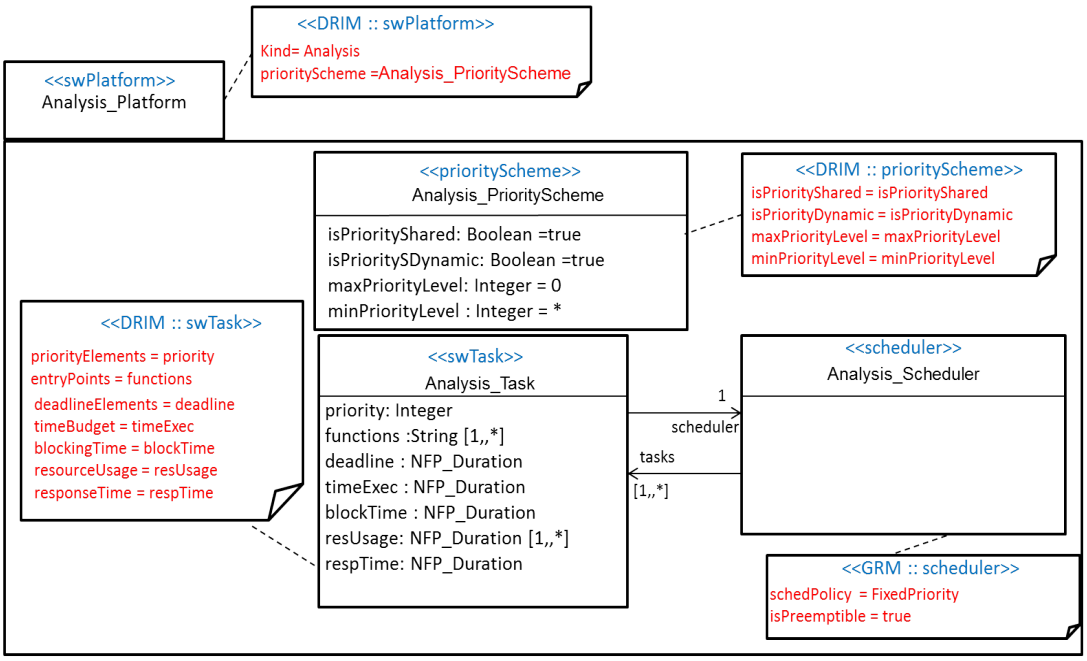


FIGURE 4.15 – *Exemple d'application de la huitième et neuvième règle pour une plateforme d'analyse*

priorityScheme du même stéréotype et qui correspond à *MicroC_PriorityScheme* pour MicroC-OS/II (respectivement à *Analysis_PriorityScheme* pour la plateforme d'analyse).

4.3.2 Les règles de modélisation des applications temps réel

Dans le processus DRIM, l'application est décrite à deux niveaux d'abstraction : au niveau conception au travers la définition d'un modèle de conception de celle-ci et au niveau implémentation au travers la génération d'un modèle de cette dernière spécifique à un RTOS. Ainsi, nous identifions dans cette partie les règles méthodologiques pour la description d'une application temps réel à ces deux niveaux d'abstraction dans le processus DRIM.

– **Règle A1 :** *Le modèle de conception d'une application est représenté par un* Paquetage *UML annoté par «GaAnalyisContext»*

Le stéréotype «GaAnalyisContext» permet, d'une part, de référencer l'application pour laquelle nous cherchons à générer le modèle de conception au travers la propriété *workload*. D'autre part, il permet de référencer la plateforme sur laquelle se base les analyses au travers la propriété *platform*.

La mise en œuvre de cette règle est donnée dans la Figure 4.16. Ainsi, le paquetage appelé *Design_Model* qui représente le modèle de conception de l'application est annoté par le stéréotype «GaAnalysisContext». La propriété *workload* de ce stéréotype correspond à la description comportementale de l'application (sous forme d'un graphe d'activation) stéréo-

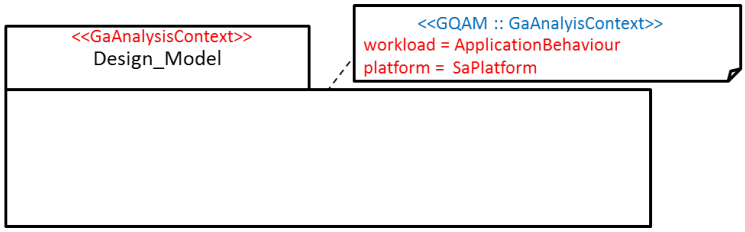


FIGURE 4.16 – *Exemple d'application de la première règle pour créer un modèle de conception de l'application*

typée par «gaWorkLoadBehavior» et annotée par les propriétés temporelles nécessaires pour la génération d'un modèle de conception temps réel. Tandis que le propriété *platform* détermine la plateforme d'analyse stéréotypée par «gaResourcesPlatform» qui correspond à une description de l'architecture matérielle de la plateforme cible (nœuds d'exécution) et d'une configuration de la plateforme logicielle abstraite d'analyse (par exemple la plateforme Analysis_Platform représentée précédemment).

– **Règle A2 :** *Le modèle d'implémentation d'une application est représenté par un paquetage UML annoté par «implementationContext» du profil DRIM.*

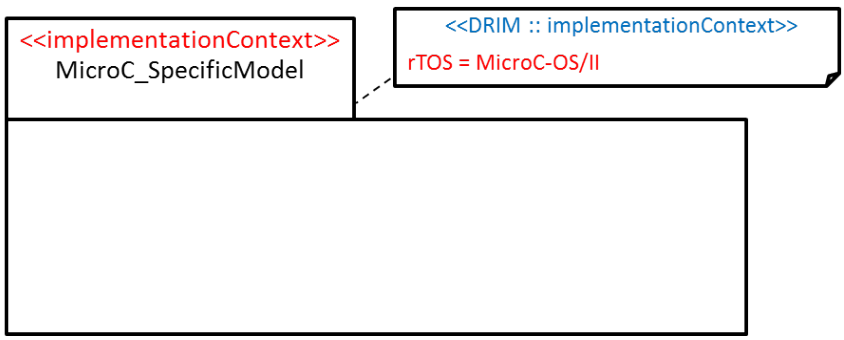


FIGURE 4.17 – *Exemple d'application de la deuxième règle pour créer un modèle de l'application spécifique au noyau MicroC-OS/II*

La Figure 4.17 montre l'application de la deuxième règle pour la création d'un modèle d'une application temps réel spécifique au noyau MicroC-OS/II. Ainsi, un paquetage nommé *MicroC-OS/II_SpecificModel* est créé. Ce paquetage est annoté par le stéréotype «implementationContext» du profil DRIM. Ce stéréotype référence ainsi le RTOS MicroC-OS/II (au travers son *tagged value rTOS*) pour mentionner que ce contexte d'implémentation est spécifique à ce RTOS.

– **Règle A3 :** *La création des modèles de l'application est basée sur la relation instance/type. Ainsi, chaque élément du modèle de l'application est une instance* (InstanceSpecification) *UML typée par un concept de la plateforme considérée.*

La Figure 4.18 illustre un exemple d'application de cette règle pour une application constituée de deux tâches. Les deux tâches de cette application sont représentées par deux instances appelées *task1* et *task2* dans les modèles de celle-ci. Le modèle de conception représente une description de l'application qui se base sur une plateforme abstraite d'analyse. Ainsi, les instances représentants les tâches, dans le modèle de conception, sont typées par le concept *Analysis_task* du modèle de la plateforme d'analyse. Le modèle d'implémentation représente une description de l'application spécifique à un noyau temps réel. Ainsi, les instances représentants les tâches, dans le modèle d'implémentation, sont typées par le concept *MicroC_task* du modèle du noyau *MicroC-OS/II*.

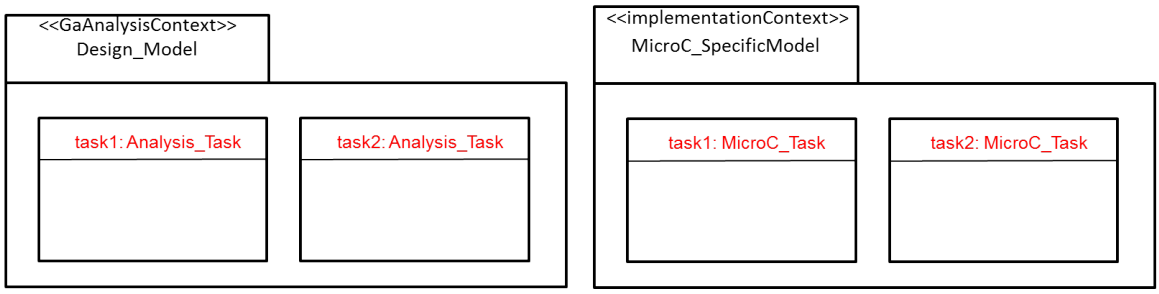


FIGURE 4.18 – *Exemple d'application de la troisième règle pour créer un modèle de conception et un modèle d'implémentation d'une application*

– **Règle A4 :** *Les valeurs des propriétés de la ressource logicielle qui représente le type d'une instance sont représentées par des* Slots *UML au niveau de l'instance (*InstanceSpecification*)de la ressource considérée.*

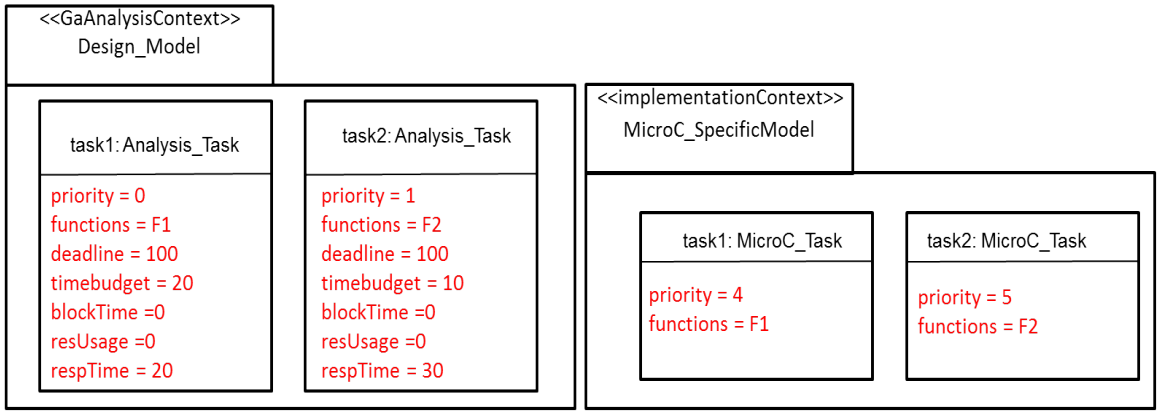


FIGURE 4.19 – *Exemple d'application de la quatrième règle pour créer un modèle de conception et un modèle d'implémentation d'une application.*

Dans la Figure 4.19, les propriétés des ressources qui caractérisent l'application (qui n'ont pas une valeur par défaut au niveau du modèle de la plateforme) auront une valeur pour

chaque instance de cette ressource. Par exemple, les deux tâches *task1* et *task2* du modèle de conception (respectivement du modèle spécifique à MicroC-OS/II) définissent toutes les propriétés de la ressource *Analysis_Task* (respectivement de la ressource *MicroC_Task*).

Synthèse

Dans cette partie, nous avons expliqué les heuristiques de description des modèles du processus DRIM en se basant sur le profil DRIM. Il est à noter que tous les algorithmes que nous allons présenter dans le prochain chapitre (pour décrire une mise en œuvre du processus DRIM) et qui auront comme entrées un ou/plusieurs de ces modèles, suppose que ces règles sont systématiquement respectées.

4.4 Conclusion

Dans ce chapitre, nous avons présenté une méthodologie pour la description des modèles des plateformes d'exécution logicielles, appelées plateformes logicielles, du processus DRIM. Pour cela, nous avons commencé par définir le profil DRIM qui est une extension du profil MARTE et qui introduit les artéfacts nécessaires pour la description des différents modèles. Ensuite, en nous basant sur ce profil, nous avons donné les heuristiques de modélisation à savoir les heuristiques de description des modèles applicatifs : modèle de conception et le modèle d'implémentation et les heuristiques de description des modèles de plateformes logicielles : modèle de plateforme abstraite d'analyse et le modèle du RTOS.

Dans le chapitre suivant, nous proposons une mise en œuvre du processus DRIM permettant d'assurer le déploiement des applications temps-réel visées dans cette étude (i.e. des applications temps-réel s'exécutant sur une architecture mono-processeur et présentant des tâches qui ne peuvent être dépendantes que par partage de ressources) sur différentes plateformes logicielles (RTOS).

Chapitre 5

Mise en œuvre du processus DRIM

Ce chapitre décrit une mise en œuvre du processus DRIM. L'objectif de cette mise en œuvre est de détailler les techniques permettant de guider le déploiement des applications visées dans cette étude (i.e. des applications temps-réel s'exécutant sur une architecture mono-processeur et présentant des tâches qui ne peuvent être dépendantes qu'en partageant des données en exclusion mutuelle).

5.1 Introduction

Dans les chapitres précédents nous avons introduit le processus DRIM, qui intègre un ensemble d'étapes intermédiaires entre la phase de conception et la phase d'implémentation du flot IDM dans le but d'assurer le déploiement multiplateforme des applications temps réel. Par la suite, un langage pour la description des modèles du processus DRIM (le profil DRIM) a été définit et un ensemble de règles expliquant la méthodologie à suivre pour la création de ces différents modèles ont été également exposées. Dans ce chapitre, nous proposons une mise en œuvre du processus DRIM, au travers l'introduction de quelques exemples de tests, de stratégies de portage et de patrons permettant de guider le déploiement des applications visées dans cette étude (i.e. des applications temps-réel s'exécutant sur une architecture mono-processeur et présentant des tâches qui ne peuvent être dépendantes que par partage de ressources).

Ainsi, ce chapitre est structuré comme suit. La première partie explique la phase d'évaluation de faisabilité et présente quelques exemples de tests. Dans la deuxième partie, nous expliquons les techniques de génération et de validation des modèles d'implémentation. La troisième partie introduit et explique quatre exemples de patrons sur lesquels se base l'étape de refactoring, avant de conclure dans la quatrième partie ce chapitre.

5.2 Évaluation de faisabilité du déploiement

Au niveau conception, la description d'une application temps réel dans le but de trouver une réalisation qui satisfait les contraintes de temps de celle-ci (en faisant appel à des techniques d'analyse), nécessite une considération de certains choix (c'est à dire des choix architecturaux et des hypothèses liées à la plateforme logicielle sous-jacente). A ce niveau, les choix du concepteur sont indépendants de n'importe quel RTOS. Par contre, au niveau implémentation, la description d'une application subit des limitations engendrées par les ressources offertes par le RTOS considéré. Ainsi, certain choix réalisés au niveau conception peuvent être non-implémentables.

Partant de ce constat, la mise en place d'une phase pour l'évaluation de la faisabilité du déploiement du modèle de conception en entrée sur le RTOS considéré s'avère d'une importance majeure. L'objectif de cette phase est de détecter les problèmes de déploiement en se basant sur un ensemble de tests de faisabilité. En cas de problème, cette phase génère une erreur au concepteur pour signaler son origine. Nous présentons maintenant des tests de faisabilité pour le cas monoprocesseur. Ces tests sont basés sur les notations présentées dans le méta-modèle DRIM pour la description des plateformes logicielles.

5.2.1 Test de l'ordonnanceur

La vérification des contraintes temporelle d'une application nécessite la détermination de l'ordonnanceur sur lequel celle-ci se base. Le choix de l'ordonnanceur a un im-

pact majeur sur la façon d'interpréter le modèle de l'application et par la suite sur les résultats d'analyse effectuée. En effet, la technique d'analyse à appliquer dépend fortement de ce choix d'ordonnanceur [19][49] ce qui se traduit par des résultats d'analyse différents. De même, si le modèle de l'application spécifique au RTOS cible utilise un ordonnanceur différent de celui utilisé au niveau conception pour vérifier les contraintes de temps, ces contraintes ne seront pas forcément respectées à l'implémentation.

Algorithme 1 : Test de l'ordonnanceur

Entrées :
$swPlatform_{analysis}$: un modèle qui décrit une configuration d'une plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
Résultat :
Verdict S= {E, NP} ; E : Erreur, NP : Pas de Problème
Données :
$Scheduler_{analysis}$: l'ordonnanceur fourni par la plateforme abstraite d'analyse
$Scheduler_{RTOS}$: l'ordonnanceur fourni par le RTOS cible
$SchedPolicy_{analysis}$: la politique d'ordonnancement fixée par le concepteur pour l'analyse
$Mode_{analysis}$: le mode de l'ordonnanceur fixé par le concepteur pour l'analyse
$SchedPolicies_{RTOS}$: La liste des politiques d'ordonnancement fournies par le RTOS cible
$Mode_{RTOS}$: la liste des modes fournis par le RTOS cible
Initialisations :
$S \leftarrow E$;
début
    $Scheduler_{analysis} = swPlatform_{analysis}$.getscheduler() ;
    $Scheduler_{RTOS} = swPlatform_{RTOS}$.getscheduler() ;
    $SchedPolicy_{analysis} = Scheduler_{analysis}$.getschedPolicy() ;
    $Mode_{analysis} = Scheduler_{analysis}$.getmode() ;
    $SchedPolicies_{RTOS} = Scheduler_{RTOS}$.getschedPolicy() ;
    $Mode_{RTOS} = Scheduler_{RTOS}$.getmode() ;
    si $Mode_{RTOS}$.size()==1 **alors**
        si $Mode_{RTOS}$!= $Mode_{analysis}$ **alors** **Retourner** S ;
    pour *i* **allant de** *0* **à** $SchedPolicies_{RTOS}$.size() **faire**
        si $SchedPolicies_{RTOS}$.get(i)==$SchedPolicy_{analysis}$ **alors**
            $S \leftarrow NP$;
            Retourner S ;
    Retourner S ;

Ainsi, l'intérêt de ce test est de vérifier si l'ordonnanceur utilisé au niveau conception (à des fins d'analyse) est fourni par le RTOS cible. Dans le cas échéant, le déploiement est faisable et nous pouvons ainsi générer un modèle de l'application spécifique à ce RTOS qui sera basé sur le même ordonnanceur utilisé au niveau conception. Dans le cas contraire, ce test génère une erreur pour signaler que le modèle de conception en entrée n'est pas implé-

mentable sur le RTOS considéré. Une description algorithmique de ce test est donnée par l'algorithme 1. Ce test définit comme entrées un modèle de la plateforme abstraite utilisée pour l'analyse (i.e. une configuration de cette dernière) et le modèle du RTOS cible et comme sortie un verdict qui peut être égale à *E* (c'est dire une erreur) si le RTOS considéré ne fournit pas le même ordonnanceur que celui utilisé dans la plateforme d'analyse et à *NP* (pas de problème) dans le cas contraire.

Comme nous l'avons montré dans 4.2.1, un ordonnanceur est caractérisé par les politiques d'ordonnancement qu'il supporte et son mode. Pour cela, cet algorithme commence par déterminer la politique d'ordonnancement utilisée au niveau conception pour l'analyse ainsi que le mode considéré. Ensuite, il détermine les politiques d'ordonnancement et les modes fournis par le RTOS cible. Comme nous l'avons signalé, une configuration d'une plateforme abstraite d'analyse ne peut définir qu'un seul mode et qu'une seule politique, cependant, un RTOS peut fournir plus qu'un mode et plus qu'une politique. Ainsi, si le même ordonnanceur existe, cet algorithme signale qu'il n'y a pas de problème. Sinon, une erreur est générée pour informer le concepteur que ce type d'ordonnanceur n'est pas fourni par le RTOS considéré.

5.2.2 Test des ressources partagées

Au niveau conception, la vérification des contraintes temporelles d'une application constituée de tâches qui partagent des ressources, nécessite la définition des choix d'implémentation des différentes ressources partagées (c'est-à-dire des sections critiques). La définition de ces choix revient à déterminer le mécanisme d'accès à la section critique (par exemple sémaphore, mutex, etc.), la politique de la file qui permet de gérer l'accès à la section critique (par exemple FIFO, basée sur les priorités, etc.) et le protocole de synchronisation utilisé (par exemple PCP, PIP, etc.). En effet, la technique d'analyse à appliquer dépend fortement de ces choix. Plus précisément, la technique utilisée pour le calcul du temps de blocage d'une tâche dépend du protocole de synchronisation utilisé, ce qui se traduit par des résultats d'analyse différents. Parallèlement, si le modèle de l'application spécifique au RTOS cible utilise un protocole de synchronisation pour protéger l'accès à une ressource partagée différent de celui utilisé au niveau conception pour vérifier les contraintes temps réel de l'application, ces contraintes ne seront pas forcement respectées à l'implémentation.

Ainsi, l'intérêt de ce test est de vérifier si le RTOS cible offre la possibilité d'implémenter une ressource partagée avec le même protocole de synchronisation défini au niveau conception (à des fins d'analyse). Dans le cas échéant, le déploiement est faisable. Dans le cas contraire, ce test génère une erreur pour signaler que le modèle de conception en entrée n'est pas implémentable sur le RTOS considéré. Une description algorithmique de ce test est donnée par l'algorithme 2.

Ce test définit comme entrées le modèle de la plateforme abstraite d'analyse (i.e. une configuration de cette dernière) et le modèle du RTOS cible et comme sortie un verdict qui peut être égal à *E* (c'est dire une erreur) si le RTOS considéré ne fournit pas le

même protocole de synchronisation utilisé pour implémenter une ressource partagée au niveau conception à des fins d'analyse et à *NP*(pas de problème) dans le cas contraire.

Algorithme 2 : Test des ressources partagées

Entrées :
$swPlatform_{analysis}$: un modèle qui décrit une configuration d'une plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
Résultat :
Verdict S= {E, NP} ; E : Erreur, NP : Pas de Problème
Données :
$Resource_{analysis}$: l'implémentation d'une ressource partagée considérée au niveau conception à des fins d'analyse
$Resources_{RTOS}$: la liste des implémentations possibles d'une ressource partagée fournies par le RTOS cible.
$AccessProtocol_{analysis}$: le protocole de synchronisation choisi au niveau conception pour implémenter une ressource partagée à des fins d'analyse
$AccessProtocols_{RTOS}$: La liste des protocoles de synchronisation supportés par le RTOS cible
Initialisations :
$S \leftarrow E$;
début
    $Resource_{analysis} = swPlatform_{analysis}$.getresource()
    $Resources_{RTOS}= swPlatform_{RTOS}$.getresource() ;
    $AccessProtocol_{analysis} = Resource_{analysis}$.getaccessProtocol() ;
    pour *i* **allant de** *0* **à** $Resources_{RTOS}.size()$ **faire**
        $AccessProtocols_{RTOS}$.set(i)= $Resources_{RTOS}$.get(i).getaccessProtocol() ;
    pour *i* **allant de** *0* **à** $AccessProtocols_{RTOS}.size()$ **faire**
        si $AccessProtocols_{RTOS}.get(i)== AccessProtocol_{analysis}$ **alors**
            $S \leftarrow NP$;
            Retourner S ;
    Retourner S ;

Comme nous l'avons expliqué dans 4.2.1, pour une configuration donnée d'une plateforme abstraite d'analyse, une seule implémentation d'une ressource partagée est possible. En effet, nous supposons que toutes les ressources partagées dans le modèle de tâches sont implémentées en utilisant les mêmes choix. Ainsi, la variable $Resource_{analysis}$ dans l'algorithme 2 permet de capturer cette implémentation. Par contre, un RTOS peut fournir une ou plusieurs implémentations possibles d'une ressource partagée. Ces implémentations sont capturées dans la liste $Resources_{RTOS}$ dans cet algorithme. Ensuite, l'algorithme 2 détermine le protocole de synchronisation utilisé pour l'analyse et le/les protocoles de synchronisation fournis par le RTOS cible. Si le même protocole existe, cet algorithme signale qu'il n'y a pas de problème. Sinon, une erreur est générée pour informer le concepteur que ce type d'implémentation de la ressource partagée n'est pas fourni par le RTOS considéré.

5.2.3 Identification des tests pour la notion de priorité

Dans le cas d'un ordonnanceur à base de priorité, l'affectation des priorités aux différentes tâches qui constituent l'application au niveau conception est une étape importante dans la description de celle-ci. En effet, au niveau conception, l'objectif majeur du concepteur est de trouver une configuration (une affectation possible des priorités aux tâches) qui satisfait les contraintes de temps de l'application. Dans ce qui suit, nous identifions trois tests à vérifier pour confirmer qu'une configuration de tâches est implémentable sur le RTOS considéré.

5.2.3.1 Test des niveaux de priorité distincts

Durant l'étape d'affectation des priorités aux tâches, le concepteur considère qu'il dispose d'un nombre illimité des niveaux de priorité distincts ce qui n'est pas toujours vrai à l'implémentation. En effet, la majorité de RTOS proposent un nombre borné des niveaux de priorité distincts qui varie selon leurs spécifications. Pour les applications de grande échelle, le nombre des niveaux de priorité distincts utilisés au niveau conception pour décrire l'application, peut dépasser le nombre autorisé par le RTOS cible. Dans le cas échéant, le modèle de conception en entrée n'est pas implémentable sur le RTOS considéré. D'autre part, au niveau implémentation, d'autres contraintes peuvent être considérées. Par exemple, pour des raisons d'extensibilité, le concepteur peut choisir de limiter le nombre des niveaux de priorité disponibles pour son application afin d'autoriser l'ajout d'autres fonctionnalités applicatives (tâches) ultérieurement dans la même plateforme.

Ainsi, l'intérêt de ce test est de vérifier que le nombre de niveaux de priorité distincts utilisé au niveau conception est autorisé par le RTOS cible pour l'application considérée. Algorithme 3 donne une description algorithmique de ce test.

Ce test définit comme entrées le modèle de conception de l'application ($M_{conception}$), le modèle du RTOS ($swPlatform_{RTOS}$) cible et le degré d'extensibilité qui représente une contrainte d'implémentation supplémentaire qui limite le nombre de niveaux de priorité distincts pour l'application considérée. Ainsi, durant le déploiement, le concepteur peut selon ses besoins choisir de donner une valeur à ce paramètre. Cette valeur représente le nombre de niveaux de priorité distincts réservé pour l'application considérée au niveau implémentation. Ce test génère comme sortie un verdict qui peut être égale à E (c'est dire une erreur) dans le cas où le nombre de priorités distinctes utilisé dans le modèle de conception dépasse le nombre autorisé pour celle-ci au niveau implémentation. Ce test signale qu'il n'y a pas de problème (NP) dans le cas contraire.

Pour cela, cet algorithme commence par déterminer le nombre des niveaux de priorité distincts utilisé au niveau conception($NbrLevels_{conception}$). Ensuite, il procède à une comparaison de ce nombre avec le degré d'extensibilité s'il est *non nul*. Dans le cas où le degré d'extensibilité est *nul*, cet algorithme compare ce nombre avec le nombre des niveaux de priorité distincts autorisé par le RTOS. En effet, la détermination de ce nombre est basée sur son modèle. Comme nous l'avons montré dans le chapitre précédent, nous caractérisons

une plateforme logicielle (en particulier un RTOS) par un schéma de priorité. Ce schéma de priorité définit en particulier des propriétés qui représentent le niveau maximal et le niveau minimal de priorité autorisés par la plateforme logicielle considérée. Les valeurs de ces deux propriétés (au niveau du modèle du RTOS) représentent respectivement les valeurs des variables *MaxPriorityLevel* et *MinPriorityLevel* dans l'algorithme 3. En se basant sur les résultats de ces comparaisons, cet algorithme produit la sortie appropriée.

Algorithme 3 : Test du nombre des niveaux de priorité distincts

Entrées :
$M_{conception}$: le modèle de conception de l'application basé sur une configuration de la plateforme abstraite d'analyse ($swPlatform_{analysis}$)
$swPlatform_{RTOS}$: le modèle du RTOS cible
Ext_Degree : Le nombre des niveaux de priorité distincts réservé pour l'application au niveau implémentation
Résultat :
Verdict S= {E, NP} ; E : Erreur, NP : Pas de Problème
Données :
$NbrLevels_{conception}$: le nombre des niveaux de priorité distincts utilisé pour décrire l'application au niveau conception
$NbrLevels_{RTOS}$: le nombre des niveaux de priorité distincts autorisé par le RTOS cible
$priorityScheme_{RTOS}$: le schéma de priorité du RTOS cible
MaxPriorityLevel : le niveau maximal de priorité autorisé par le RTOS cible
MinPriorityLevel : le niveau minimal de priorité autorisé par le RTOS cible
Initialisations :
$S \leftarrow NP$;
début
  $NbrLevels_{conception}$ = getNumberPriorityLevels($M_{conception}$) ;
  $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
  MaxPriorityLevel = $priorityScheme_{RTOS}$.getmaxPriorityLevel() ;
  MinPriorityLevel = $priorityScheme_{RTOS}$.getminPriorityLevel() ;
  $NbrLevels_{RTOS}$ = | *MaxPriorityLevel* - *MinPriorityLevel* | ;
  si *Ext_Degree* != **null alors**
    si $NbrLevels_{conception}$ > *Ext_Degree* **alors**
      $S \leftarrow E$;
      Retourner S ;
  sinon
    si $NbrLevels_{conception}$> $NbrLevels_{RTOS}$ **alors**
      $S \leftarrow E$;
      Retourner S ;
  Retourner S ;

5.2.3.2 Test des niveaux de priorité égaux

Durant l'étape d'affectation des priorités aux tâches, le concepteur peut choisir de donner un même niveau de priorité à plusieurs tâches. Cependant, ceci n'est pas toujours possible à l'implémentation. En effet, certains RTOS n'offrent pas la possibilité de partager les niveaux

de priorité entre tâches. Un exemple de cette famille de RTOS est MicroC-OS/II [10].

Ainsi, dans le cas où le modèle de conception en entrée présente des tâches qui partagent le même niveau de priorité, l'intérêt de ce test est de vérifier si le RTOS cible permet d'exécuter des tâches avec des niveaux de priorité égaux. Algorithme 4 présente une description algorithmique de ce test.

Algorithme 4 : Test des niveaux de priorité égaux

Entrées :
$M_{conception}$: le modèle de conception de l'application basé sur une configuration de la plateforme abstraite d'analyse ($swPlatform_{analysis}$)
$swPlatform_{RTOS}$: le modèle du RTOS cible
Résultat :
Verdict S= {E, NP} ; E : Erreur, NP : Pas de Problème
Données :
$isPriorityShared_{conception}$: une variable booléenne pour indiquer si le modèle de conception en entrée présente des tâches qui partagent le même niveau de priorité
$priorityScheme_{RTOS}$: le schéma de priorité du RTOS cible
$isPriorityShared_{RTOS}$: Une variable booléenne pour indiquer si le cible RTOS supporte le partage des niveaux de priorité entre tâches
Initialisations :
$S \leftarrow NP$;
début
    $isPriorityShared_{conception}$ = getExistShared($M_{conception}$) ;
    $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
    $isPriorityShared_{RTOS}$ = $priorityScheme_{RTOS}$.getisPriorityShared() ;
    si $isPriorityShared_{conception}$ =**vrai alors**
        si $isPriorityShared_{RTOS}$ =**faux alors**
            $S \leftarrow E$;
            Retourner S ;
    Retourner S ;

Ce test définit comme entrées le modèle de conception de l'application ($M_{conception}$) et le modèle du RTOS ($swPlatform_{RTOS}$) cible et produit comme sortie un verdict qui peut être égale à E (c'est dire une erreur) dans le cas où le modèle de conception présente des tâches avec un même niveau de priorité et le RTOS cible n'autorise pas une telle situation. Ce test signale qu'il n'y a pas de problème (NP) dans le cas contraire. Cet algorithme commence par vérifier si le modèle de conception définit au moins deux tâches qui partagent un même niveau de priorité($isPriorityShared_{conception}$). Dans le cas échéant, cet algorithme vérifie si le RTOS cible supporte le partage des niveaux de priorité entre tâches en se basant sur son modèle. Si ce n'est pas le cas, cet algorithme produit une erreur. En effet, comme nous l'avons montré dans le chapitre précédent, le schéma de priorité associé à une plateforme logicielle définit une propriété *isPriorityShared* qui permet de confirmer si le RTOS supporte le partage d'un même niveau de priorité. Ainsi la valeur de cette propriété correspond exactement à la valeur de la variable ($isPriorityShared_{RTOS}$) dans l'algorithme 4.

5.2.3.3 Test de priorité dynamique

Durant l'étape d'affectation des priorités, certaines tâches peuvent être définies par plus qu'une valeur de priorité. Ceci veut dire que le concepteur suppose que les priorités de ses tâches peuvent varier en cours de l'exécution. Ainsi, pour une telle configuration, une tâche peut être définie comme étant la plus prioritaire au début de l'exécution et comme étant la moins prioritaire à la fin de l'exécution. Cependant, l'implémentation des tâches ayant des priorités dynamiques n'est pas toujours supportée par les RTOS.

Par conséquent, dans le cas où le modèle de conception présente des tâches avec des priorités dynamiques, l'intérêt de ce test est de vérifier si le RTOS cible autorise que la valeur de priorité d'une tâche change en cours de l'exécution. Algorithme 5 donne une description algorithmique de ce test.

Algorithme 5 : Test des priorités dynamiques

Entrées :
$M_{conception}$: le modèle de conception de l'application basé sur une configuration de la plateforme abstraite d'analyse ($swPlatform_{analysis}$)
$swPlatform_{RTOS}$: le modèle du RTOS cible
Résultat :
Verdict S= {E, NP} ; E : Erreur, NP : Pas de Problème
Données :
$isPriorityDynamic_{conception}$: une variable booléenne pour indiquer si le modèle de conception en entrée présente des tâches qui sont définies avec plus qu'un niveau de priorité
$priorityScheme_{RTOS}$: le schéma de priorité du RTOS cible
$isPriorityDynamic_{RTOS}$: une variable booléenne pour indiquer si le RTOS supporte la variation de priorité au cours de l'exécution
Initialisations :
$S \leftarrow NP$;
début
    $isPriorityDynamic_{conception}$ = getExistDynamic($M_{conception}$) ;
    $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
    $isPriorityDynamic_{RTOS}$ = $priorityScheme_{RTOS}$.getisPriorityDynamic() ;
    si $isPriorityDynamic_{conception}$ =**vrai alors**
        si $isPriorityDynamic_{RTOS}$ =**faux alors**
            $S \leftarrow E$;
            Retourner S ;
    Retourner S ;

Ce test définit comme entrées le modèle de conception de l'application ($M_{conception}$) et le modèle du RTOS ($swPlatform_{RTOS}$) cible et produit comme sortie un verdict qui peut être égale à E (c'est dire une erreur) dans le cas où le modèle de conception présente des tâches avec plus qu'un niveau de priorité et le RTOS cible n'autorise pas une telle situation. Ce test signale qu'il n'y a pas de problème (NP) dans le cas contraire. Cet algorithme commence par vérifier si le modèle de conception est constitué d'au moins une tâche qui est définie avec

plus qu'un niveau de priorité ($isPriorityDynamic_{conception}$). Dans le cas échéant, il vérifie si le RTOS cible supporte la variation de priorité au cours de l'exécution en se basant sur son modèle. Si ce n'est pas le cas, cet algorithme produit une erreur. En effet, comme nous l'avons montré dans le chapitre précédent, le schéma de priorité associé à une plateforme logicielle définit une propriété *isPriorityDynamic* qui permet de confirmer si le RTOS cible supporte la notion de priorité dynamique. Ainsi la valeur de cette propriété correspond exactement à la valeur de la variable ($isPriorityDynamic_{RTOS}$) dans l'algorithme 5.

Synthèse

Le tableau 5.1 donne une description des différents tests identifiés dans le but d'évaluer la faisabilité de déploiement des modèles de conception des applications temps réel visées dans cette étude sur un RTOS cible. Ainsi, à la fin de cette phase, un modèle de conception

TABLE 5.1 – Liste des tests de faisabilité du déploiement identifiés dans cette étude

Test	Description
Test de l'ordonnanceur	Vérifie si le RTOS cible fournit le même ordonnanceur que celui utilisé pour l'analyse
Test des ressources partagées	Vérifie si le RTOS cible fournit le même protocole de synchronisation utilisé pour l'implémentation d'une ressource partagée que celui utilisé pour l'analyse
Test du nombre des niveaux de priorité distincts	Vérifie si le nombre des niveaux de priorité distincts utilisé pour décrire l'application au niveau conception est autorisé par le RTOS cible pour l'application considérée
Test des niveaux de priorité égaux	Vérifie si le RTOS cible autorise le partage d'un même niveau de priorité au cas où le modèle de conception présente des tâches ayant la même priorité
Test de priorité dynamique	Vérifie si le RTOS cible autorise la variation de la valeur de priorité au cours de l'exécution au cas où le modèle de conception présente des tâches ayant plus qu'un niveau de priorité

d'une application temps réel est déclaré implémentable sur un RTOS cible *si et seulement si* tous les tests de faisabilité définis dans cette partie produit un retour positif (c'est-à-dire pas de problème *NP*). Dans ce cas, le processus DRIM passe à la phase de portage qui permet de générer le modèle de l'application spécifique à ce RTOS. Ensuite, DRIM procède à une étape de validation de ce portage. Ce scénario sera présenté en plus de détails dans la section suivante. Dans le cas où au moins un test produit une erreur, le processus DRIM passe à l'étape de refactoring dans le but de trouver une solution au problème signalé. Ce scénario sera l'objet de la section 5.4 de ce chapitre.

5.3 Génération et validation des modèles d'implémentation

Dans cette partie, nous présentons en plus de détails la phase de portage qui permet de générer le modèle d'implémentation (ou le modèle spécifique à un RTOS) d'une application temps réel à partir du modèle de conception de celle-ci. Par la suite, nous expliquons la phase de validation du portage qui permet d'évaluer la conformité du modèle d'implémentation résultant par rapport au modèle de conception de départ dans le but d'assurer le respect des contraintes de temps à l'implémentation.

5.3.1 Phase de Portage

L'objectif de cette phase est d'assurer un déploiement qui permet de générer des modèles de l'application spécifiques à des RTOS à partir du modèle de conception. Cette génération doit être en mesure de préserver la spécification de haut niveau de l'application et d'assurer le respect des contraintes de temps vérifiées au niveau conception tout en prenant en considération les caractéristiques du RTOS cible. Afin de réaliser son objectif, cette phase est basée sur la notion de *mise en correspondance* entre les ressources logicielles abstraites décrites dans le modèle de la plateforme d'analyse et les ressources logicielles d'exécution décrites dans le modèle du RTOS. Elle introduit, en plus, une étape de portage des valeurs des propriétés qui permet de traduire fidèlement le modèle de conception de l'application en un modèle d'implémentation en tenant compte des caractéristiques du RTOS cible. Nous définissons, dans ce qui suit, les règles de correspondance définies pour assurer le déploiement d'une application sur un RTOS, avant d'introduire le principe de portage et quelques techniques de portage des valeurs des propriétés à savoir les niveaux de priorité et les périodes.

5.3.1.1 Définition des règles de correspondance

La mise en correspondance entre les ressources des plateformes logicielles (abstraite d'analyse et RTOS) et les propriétés de ces ressources est assurée, d'une part, par l'utilisation d'un même langage, appelé langage pivot, pour la description des deux types de plateformes. D'autre part, cette mise en correspondance est assurée par la définition d'un ensemble de règles de correspondance. En effet, l'utilisation d'un langage pivot, permet d'identifier par stéréotypage, en se basant sur les règles de correspondance, les éléments du modèle source et les éléments correspondants dans le modèle cible.

Dans ce travail, la description de la plateforme abstraite d'analyse et du RTOS est réalisée en utilisant le même langage (le profil DRIM) qui sert comme pivot pour la mise en correspondance. Nous identifions ainsi les règles de correspondance sur lesquelles se base l'algorithme de portage (Algorithme 6).

Les règles de correspondance des ressources Nous supposons qu'une ressource R de la plateforme abstraite d'analyse **correspond à** une ressource R' du RTOS, *si et seulement si* :

- **Règle C1** : les deux ressources sont annotées par le même stéréotype du profil DRIM : pour chaque instance du modèle de départ dont le type est stéréotypé dans le modèle plateforme abstraite source, s'il existe une ressource ayant le même stéréotype appliqué dans le modèle du RTOS cible, cette ressource est instanciée.
- **Règle C2** : toutes les propriétés du stéréotype qui référencent des attributs de la ressource *R'* du RTOS, référencent aussi des attributs de la ressource *R* de la plateforme d'analyse. Si plusieurs ressources sont candidates celles qui partagent de propriétés sera élues.
- **Règle C3** : les propriétés du stéréotype appliqué à la ressource *R* ayant des valeurs par défaut ont aussi les mêmes valeurs par défaut pour la ressource *R'*.

Les règles de correspondance des propriétés des ressources L'identification des propriétés des ressources qui correspondent permet de capturer celles qui nécessitent un portage approprié de leur valeur. Ces propriétés sont en général des propriétés qui caractérisent l'application comme par exemple la priorité d'une tâche. Nous supposons ainsi qu'une propriété *p* de la ressource *R* **correspond à** une propriété *p'* de la ressource *R'*, *si et seulement si* :

- **Règle C4** : les deux propriétés (*p* et *p'*) sont référencées par la même propriété du stéréotype appliqué.
- **Règle C5** : les propriétés (*p* et *p'*) n'ont pas des valeurs par défaut au niveau la ressource (i.e. les propriétés qui ont des valeurs par défaut au niveau du modèle de la plateforme caractérisent cette plateforme et ne nécessitent pas un portage approprié de leur valeur)

La Figure 5.1 illustre un exemple de mise en correspondance en se basant sur les modèles de plateformes précédemment introduites dans 4.3 et les règles de correspondance que nous avons identifiées. Dans cette figure, la ressource *Analysis_Task* du modèle de la plateforme d'analyse correspond à la ressource *MicroC_Task* du modèle de MicroC-OS/II. En effet, la première règle de correspondance est vérifiée (Règle C1) puisque ces deux concepts partagent le stéréotype «swTask» tiré du profil DRIM. D'autre part, la deuxième règle est vérifiée (Règle C2) car les propriétés *priorityElements* et *entryPoints* de ce stéréotype référencent des attributs de *Analysis_Task* et de *MicroC_Task*. Pour la troisième règle (Règle C3), aucune propriété de ce stéréotype ne définit une valeur par défaut ainsi elle est supposé vérifiée. En ce qui concerne les propriétés, nous pouvons constaté que l'attribut *priority* de la classe *Analysis_Task* (respectivement *functions* de la classe *Analysis_Task*) correspond à l'attribut *priority* de la classe *MicroC_Task*(respectivement *functions* de la classe *MicroC_Task*). En effet, ces propriétés sont référencées par les mêmes propriétés *priorityElements* et *entryPoints* du stéréotype «swTask» et n'ont pas des valeurs par défaut, ce qui confirme que les règles C4 et C5 sont vérifiées.

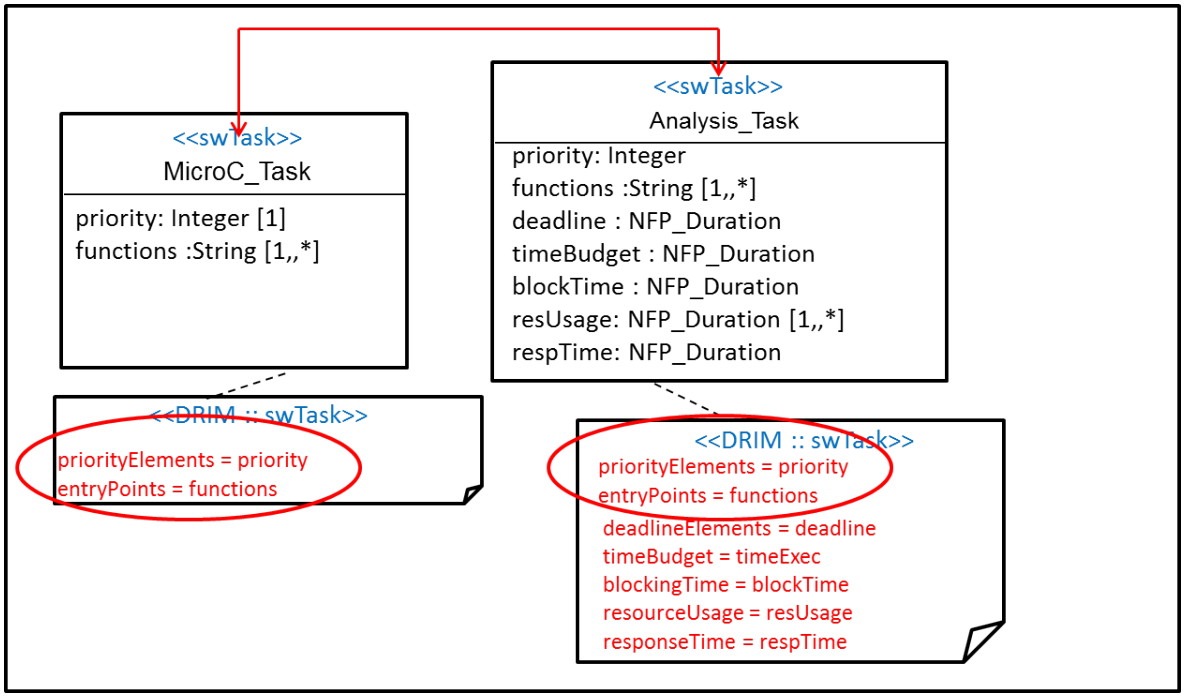


FIGURE 5.1 – *Un exemple qui illustre une correspondance entre une tâche MicroC-OS/II et une tâche d'une plateforme abstraite d'analyse*

5.3.1.2 Principe de portage

Dans ce travail, le déploiement d'une application temps réel sur un RTOS est assuré par une opération du portage de l'application d'une plateforme source (qui représente une plateforme abstraite d'analyse) à une plateforme cible (qui représente le RTOS) [86]. Cette étape de portage est illustrée par l'algorithme 6.

Cet algorithme définit comme entrées le modèle conception de l'application, un modèle qui décrit une configuration d'une plateforme abstraite d'analyse et le modèle du RTOS cible. Il produit comme sortie le modèle de l'application, représentée par son modèle en entrée, spécifique au RTOS (modèle d'implémentation). Cet algorithme est basé sur deux niveaux de portage : le portage des instances applicatives par une mise en correspondance entre leurs types et le portage des valeurs des propriétés des instances applicatives par une mise en correspondance entre les propriétés de leurs types. En effet, la mise en correspondance entre les types des instances se traduit par une mise en correspondance entre les ressources logicielles abstraites du modèle de la plateforme abstraite d'analyse et les ressources logicielles concrètes du modèle du RTOS. En outre, la mise en correspondance entre les propriétés des types des instances correspond à une mise en correspondance entre les propriétés de ces ressources.

Algorithme 6 : Algorithme de Portage

Entrées :
$M_{conception}$: le modèle de conception de l'application basé sur une configuration d'une plateforme abstraite d'analyse
$swPlatform_{analysis}$ un modèle qui décrit une configuration d'une plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
Résultat :
$M_{implementation}$: le modèle d'implémentation de l'application spécifique au RTOS considéré ($swPlatform_{RTOS}$)
Données :
$Type(i)$: le concept du modèle de plateforme qui type l'instance i
$Property(c)$: l'ensemble des propriétés du concept C de la plateforme
$Value_P(i)$: la valeur de la propriété p de l'instance i
début
  pour tout $i \in M_{conception}$ **tel que** $Type(i) \in swPlatform_{analysis}$ **faire**
    $M_{implementation}$ = ajouter(i', $Type(i')$) **tel que** $Type(i)$ **correspond à** $Type(i') \in swPlatform_{RTOS}$;
    pour tout $p \in Property(Type(i))$ **faire**
      si $\exists\, p' \in Property(Type(i'))$ **tel que** p' **correspond à** p **alors**
        $Value_P(i') = Portage_P(Value_P(i))$;
  Retourner $M_{implementation}$;

Ainsi, pour chaque instance du modèle de conception de type T qui correspond à une ressource dans la plateforme abstraite d'analyse, cet algorithme génère une instance dans le modèle d'implémentation de type T' qui correspond à une ressource dans le modèle de du RTOS. Cette génération est effectuée si et seulement si *T correspond à T'*. Une fois le type T' de l'instance du modèle d'implémentation générée, cet algorithme procède au deuxième niveau de portage ($Portage_P$) qui est le portage des valeurs des propriétés qui correspondent (c'est-à-dire les propriétés du type T' qui correspondent aux propriétés de type T). Ce deuxième niveau de portage permet de déterminer les valeurs des propriétés des instances du modèle d'implémentation à partir des valeurs des propriétés des instances du modèle de conception en tenant compte des caractéristiques du RTOS cible. Dans ce qui suit, nous nous intéressons à deux exemples de propriétés à savoir la priorité et la période d'une tâche. Nous présentons ainsi les différents algorithmes proposés pour le portage des valeurs de ces deux propriétés.

Portage des niveaux de priorités

Ce portage doit garantir que l'ordre de précédence, en termes de niveaux de priorité, des tâches au niveau implémentation correspond exactement à l'ordre spécifié au niveau conception. Pour cela, ce portage doit tenir compte de la sémantique de la priorité au niveau de la plateforme abstraite d'analyse et aussi au niveau du RTOS cible. Nous proposons ainsi différentes stratégies pour effectuer le portage des niveaux de priorités.

Portage direct des niveaux de priorités La stratégie la plus simple pour effectuer le portage des niveaux de priorités est le portage direct. Chaque entité concurrente dans le modèle d'implémentation aura la même valeur de priorité que celle spécifiée dans le modèle de conception. Dans le cas où le concepteur veut conserver exactement les mêmes valeurs de priorité au niveau implémentation, il choisit ce type de portage. Algorithme 7 donne une description algorithmique de cette stratégie de portage des niveaux de priorité. Cependant, ce portage n'est pas toujours faisable pour tous les RTOS car il peut mener à des modèles d'implémentation qui ne sont pas valides (non équivalents au modèle de conception source). Pour cette raison, avant d'effectuer ce type de portage il faut évaluer sa faisabilité afin d'éliminer les problèmes potentiels. Deux points sont à vérifier :

- L'ordre de priorité au niveau de la plateforme abstraite d'analyse et au niveau du RTOS doit être le même. Dans le cas contraire, nous ne pouvons pas appliquer un portage direct car nous aurons un ordre inversé de précédence des tâches .
- Tous les niveaux de priorité utilisés dans le modèle de conception sont autorisés par le RTOS cible. Dans le cas contraire, le portage direct ne peut pas être appliqué car le modèle d'implémentation résultant n'est pas valide pour le RTOS cible.

Algorithme 7 : Algorithme de portage direct des niveaux de priorité

Entrées :
$PriorityValue(T)$: la valeur de priorité de la tâche T dans le modèle de conception
Résultat :
$PriorityValue(T')$: La valeur de priorité de la tâche T' qui **correspond à** T au niveau implémentation
début
  $PriorityValue(T') = PriorityValue(T)$;
  Retourner $PriorityValue(T')$;

Portage linéaire des niveaux de priorité Le portage linéaire correspond à la stratégie la plus générique pour faire le portage des niveaux de priorité. Ce portage détermine la tâche la plus prioritaire dans le modèle de conception et lui affecte le niveau de priorité le plus haut du RTOS cible. Les entités concurrentes dans le modèle d'implémentation auront des niveaux de priorité consécutifs allant du niveau le plus prioritaire vers le moins prioritaire défini par le RTOS (Algorithme 8). Pour ce type de portage, nous n'avons pas de test particulier à effectuer.

L'algorithme 8 considère comme entrée la valeur de priorité d'une tâche T appartenant au modèle de conception. Cet algorithme calcule la valeur appropriée de la priorité de la tâche T' qui correspond à T au niveau implémentation. Ceci, en se basant sur le niveau maximal de priorité (*maxPriorityLevel*) et le niveau minimal *(minPriorityLevel*) autorisés par les plateformes logicielles (abstraites et RTOS) représentées dans leur modèles.

Bien que ce portage assure un modèle d'implémentation valide de point de vue priorité, il reste mal adapté pour le rajout des tâches au niveau implémentation puisque les niveaux de priorité qu'il génère sont successifs. Pour cela, nous proposons dans ce qui suit un autre

Algorithme 8 : Algorithme de portage linéaire des niveaux de priorité

Entrées :
$PriorityValue(T)$: la valeur de priorité de la tâche T dans le modèle de conception
Résultat :
$PriorityValue(T')$: La valeur de priorité de la tâche T' qui **correspond à** T au niveau implémentation
Données :
$M_{conception}$: le modèle de conception de l'application basé sur une plateforme abstraite d'analyse
$Type(e)$: le type du concept e du modèle de conception
$swPlatform_{analysis}$: le modèle de la plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
$Higher-levels(T)$: liste des niveaux de priorité les plus élevés que celui de la tâche T ;
$priorityScheme_{platform}$: le schéma de priorité de la plateforme considérée
$MaxPriorityLevel_{platform}$: le niveau maximal de priorité dans la plateforme considérée
$MinPriorityLevel_{platform}$: le niveau minimal de priorité dans la plateforme considérée
début
  $priorityScheme_{analysis}$= $swPlatform_{analysis}$.getpriorityScheme() ;
  $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
  $MaxPriorityLevel_{analysis}$ = $priorityScheme_{analysis}$.getmaxPriorityLevel() ;
  $MaxPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getmaxPriorityLevel() ;
  c = $priorityScheme_{analysis}$.getminpriorityLevel() ;
  $MinPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getminpriorityLevel() ;
  si $MaxPriorityLevel_{analysis} < MaxPriorityLevel_{analysis}$ **alors**
    pour tout $e \in M_{conception}$ **tel que** $Type(c) == Type(T)$ **faire**
      si $PriorityValue(e) < PriorityValue(T)$ **alors**
        si $PriorityValue(e) \nexists Higher-levels(T)$ **alors**
          add ($PriorityValue(e)$,$Higher-levels(T)$) ;
  sinon
    pour tout $e \in M_{conception}$ **tel que** $Type(c) == Type(T)$ **faire**
      si $PriorityValue(e) > PriorityValue(T)$ **alors**
        si $PriorityValue(e) \nexists Higher-levels(T)$ **alors**
          add ($PriorityValue(e)$,$Higher-levels(T)$) ;
  si $MaxPriorityLevel_{RTOS} < MinPriorityLevel_{RTOS}$ **alors**
    $PriorityValue(T')$ = $MaxPriorityLevel_{RTOS}$ + sizeOf($Higher-levels(T)$) ;
  sinon
    $PriorityValue(T')$ = $MaxPriorityLevel_{RTOS}$ - sizeOf($Higher-levels(T)$) ;
  Retourner $PriorityValue(T')$;

type de portage qui permet de définir un pas entre les différents niveaux de priorité à l'implémentation.

Portage avec un pas des niveaux de priorité Ce type de portage répond à l'inconvénient du portage linéaire précédent permettant ainsi de définir un pas entre les niveaux des priorités des tâches au niveau implémentation. Cette stratégie de portage est donnée par l'algorithme 9.

De même que l'algorithme précédent 8, cet algorithme prend en entrée la valeur de priorité d'une tâche T appartenant au modèle de conception. Il considère en plus la valeur du *pas* souhaité entre les niveaux de priorité à l'implémentation. Il calcule, de la même façon, la valeur appropriée de la priorité de la tâche T' correspondant à T au niveau implémentation en se basant sur les mêmes informations plateforme (*maxPriorityLevel* et *minPriorityLevel*) et en tenant compte du pas souhaité. Cependant, un test supplémentaire doit être considéré pour ce type de portage afin de s'assurer qu'il garantit bien un modèle d'implémentation valide pour le modèle de conception en entrée, le RTOS cible et le pas choisi. Ce test est donné par l'algorithme 10.

Algorithme 10 : Algorithme d'évaluation de faisabilité du portage avec un pas

Entrées :
$M_{conception}$: le modèle de conception de l'application basé sur une plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
pas : le pas souhaité entre les niveaux de priorité à l'implémentation
Résultat :
isPossible : variable booléenne qui sera égale à **vrai** si le portage direct est faisable
Données :
$priorityScheme_{RTOS}$:le schéma de priorité du RTOS cible
$MaxPriorityLevel_{RTOS}$:le niveau maximal de priorité dans le RTOS cible
$MinPriorityLevel_{RTOS}$:le niveau minimal de priorité dans le RTOS cible
$Nbr - Levels$: le nombre des niveaux de priorité distincts utilisés dans le modèle de conception
Initialisations :
$isPossible \leftarrow$ **vrai** ;
début
  $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
  $MaxPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getmaxPriorityLevel() ;
  $MinPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getminpriorityLevel() ;
  $Nbr - Levels$ = getPriorityLevelsNumber($M_{conception}$) ;
  si $(\lfloor \frac{|MaxPriorityLevel_{RTOS} - MinPriorityLevel_{RTOS}|}{pas} \rfloor + 1) < Nbr - Levels$ **alors**
    $isPossible \leftarrow$ **faux** ;
  Retourner isPossible ;

L'objectif de ce test est de vérifier si, en utilisant le pas considéré (*Pas*) pour porter les valeurs de priorités des différentes tâches constituant le modèle de conception source

Algorithme 9 : Algorithme de portage avec un pas des niveaux de priorité

Entrées :
$PriorityValue(T)$: la valeur de priorité de la tâche T dans le modèle de conception
pas : le pas souhaité entre deux niveaux de priorité à l'implémentation
Résultat :
$PriorityValue(T')$: La valeur de priorité de la tâche T' qui **correspond à** T au niveau implémentation
Données :
$M_{conception}$: le modèle de conception de l'application basé sur une plateforme abstraite d'analyse
$Type(e)$: le type du concept e du modèle de conception
$swPlatform_{analysis}$: le modèle de la plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
$Higher-levels(T)$: liste des niveaux de priorité les plus élevés que celui de la tâche T ;
$priorityScheme_{platform}$: le schéma de priorité de la plateforme considérée
$MaxPriorityLevel_{platform}$: le niveau maximal de priorité dans la plateforme considérée
$MinPriorityLevel_{platform}$: le niveau minimal de priorité dans la plateforme considérée
début
  $priorityScheme_{analysis}$= $swPlatform_{analysis}$.getpriorityScheme() ;
  $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
  $MaxPriorityLevel_{analysis}$ = $priorityScheme_{analysis}$.getmaxPriorityLevel() ;
  $MaxPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getmaxPriorityLevel() ;
  $MinPriorityLevel_{analysis}$ = $priorityScheme_{analysis}$.getminpriorityLevel() ;
  $MinPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getminpriorityLevel() ;
  si $MaxPriorityLevel_{analysis} < MaxPriorityLevel_{analysis}$ **alors**
    pour tout $e \in M_{conception}$ **tel que** $Type(c) == Type(T)$ **faire**
      si $PriorityValue(e) < PriorityValue(T)$ **alors**
        si $PriorityValue(e) \nexists Higher-levels(T)$ **alors**
          add ($PriorityValue(e)$,$Higher-levels(T)$) ;
  sinon
    pour tout $e \in M_{conception}$ **tel que** $Type(c) == Type(T)$ **faire**
      si $PriorityValue(e) > PriorityValue(T)$ **alors**
        si $PriorityValue(e) \nexists Higher-levels(T)$ **alors**
          add ($PriorityValue(e)$,$Higher-levels(T)$) ;
  si $MaxPriorityLevel_{RTOS} < MinPriorityLevel_{RTOS}$ **alors**
    $PriorityValue(T') = MaxPriorityLevel_{RTOS}$ + pas * sizeOf($Higher-levels(T)$) ;
  sinon
    $PriorityValue(T') = MaxPriorityLevel_{RTOS}$ - pas * sizeOf($Higher-levels(T)$) ;
  Retourner $PriorityValue(T')$;

($M_{conception}$) sur la plateforme cible ($swPlatform_{RTOS}$), nous ne risquons pas d'avoir des niveaux de priorité au-delà de l'intervalle autorisé par le RTOS.

Portage proportionnel des niveaux de priorité Comme le portage avec un pas, ce type de portage permet également d'avoir des niveaux de priorité bien répartis à l'implémentation. Seulement, l'avantage de cette stratégie par rapport au portage avec un pas, est que le pas est calculé en fonction des niveaux fournis par le RTOS (ce n'est pas un paramètre d'entrée pour l'algorithme de portage) ce qui permet d'éviter le test de vérification (Algorithme 10). Cependant, ce type de portage n'est pas possible si le RTOS cible ne fournit pas une borne supérieure des niveaux de priorité comme par exemple AUTOSAR [11].

Comme les autres algorithmes de portage de niveaux de priorité, cet algorithme (Algorithme 11 calcule la valeur appropriée du niveau de priorité pour la tâche considérée en entrée en se basant sur le niveau maximal et le niveau minimal de priorité autorisés par les plateformes logicielles (abstraite et RTOS).

La complexité de tous les algorithmes présentés dans cette partie est linéaire *O(n)* (avec *n* représente le nombre de concepts dans le modèle de conception en entrée), à l'exception de l'algorithme de portage direct (Algorithme 7) dont la complexité est constante (*O(1)*). En termes de terminaison, tous ces algorithmes présentent des boucles dont le nombre de répétions est connu à l'avance (*n*).

Portage des valeurs des périodes

L'objectif de ce portage est d'assurer qu'une tâche *T'* au niveau implémentation qui correspond à une tâche *T* au niveau conception est activée avec le même taux que celle-ci. Au niveau des RTOS, le temps est géré en *clock tick*. La valeur du tick de l'horloge est une caractéristique du RTOS qui doit être spécifiée dans son modèle. Par conséquent, un portage approprié des valeurs des périodes doit tenir compte de la valeur du *tick* du RTOS cible afin de préserver les taux d'activation des tâches à l'implémentation. Algorithme 12 présente une description du mécanisme de portage des valeurs des périodes.

Cet algorithme considère comme entrée la valeur de la période de la tâche du modèle de conception pour laquelle nous voulons déterminer la valeur appropriée au niveau implémentation. En se basant sur le taux d'activation de la tâche source au niveau conception (*PeriodValue(T)*) d'une part et la valeur du tick décrite dans le modèle du RTOS cible (*TickValue*) d'autre part, cet algorithme détermine le taux d'activation de la tâche *T'* (*PeriodValue(T')*) qui correspond à *T* à l'implémentation en *tick*.

La complexité de l'algorithme 12 est indépendante du taille des données en entrées, ainsi elle est considérée comme constante (*O(1)*). De plus, cet algorithme ne présente aucune boucle ainsi sa terminaison est garantie.

Algorithme 11 : Algorithme de portage proportionnel des niveaux de priorité

Entrées :
$PriorityValue(T)$: la valeur de priorité de la tâche T dans le modèle de conception
Résultat :
$PriorityValue(T')$: La valeur de priorité de la tâche T' qui **correspond à** T au niveau implémentation
Données :
$M_{conception}$: le modèle de conception de l'application basé sur une plateforme abstraite d'analyse
$Type(e)$: le type du concept e du modèle de conception
$swPlatform_{analysis}$: le modèle de la plateforme abstraite d'analyse
$swPlatform_{RTOS}$: le modèle du RTOS cible
$Higher-levels(T)$: liste des niveaux de priorité les plus élevés que celui de la tâche T ;
$priorityScheme_{platform}$: le schéma de priorité de la plateforme considérée
$MaxPriorityLevel_{platform}$: le niveau maximal de priorité dans la plateforme considérée
$MinPriorityLevel_{platform}$: le niveau minimal de priorité dans la plateforme considérée
$Nbr-Levels$: le nombre des niveaux de priorité distincts utilisés dans le modèle de conception
pas : le pas calculé en fonction des des niveaux maximal et minimal de priorité autorisés par le RTOS et le nombre des niveaux de priorité distincts utilisé dans le modèle de conception
début
  $priorityScheme_{analysis}$= $swPlatform_{analysis}$.getpriorityScheme() ;
  $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
  $MaxPriorityLevel_{analysis}$ = $priorityScheme_{analysis}$.getmaxPriorityLevel() ;
  $MaxPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getmaxPriorityLevel() ;
  $MinPriorityLevel_{analysis}$= $priorityScheme_{analysis}$.getminpriorityLevel() ;
  $MinPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getminpriorityLevel() ;
  $Nbr-Levels$ = getPriorityLevelsNumber($M_{conception}$) ;
  $pas = \lfloor \frac{|MaxPriorityLevel_{RTOS} - MinPriorityLevel_{RTOS}|}{Nbr-Levels} \rfloor - 1$
  si $MaxPriorityLevel_{analysis} < MaxPriorityLevel_{analysis}$ **alors**
    pour tout $e \in M_{conception}$ **tel que** $Type(c) == Type(T)$ **faire**
      si $PriorityValue(e) < PriorityValue(T)$ **alors**
        si $PriorityValue(e) \nexists Higher-levels(T)$ **alors**
          add ($PriorityValue(e)$,$Higher-levels(T)$) ;
  sinon
    pour tout $e \in M_{conception}$ **tel que** $Type(c) == Type(T)$ **faire**
      si $PriorityValue(e) > PriorityValue(T)$ **alors**
        si $PriorityValue(e) \nexists Higher-levels(T)$ **alors**
          add ($PriorityValue(e)$,$Higher-levels(T)$) ;
  si $MaxPriorityLevel_{RTOS} < MinPriorityLevel_{RTOS}$ **alors**
    $PriorityValue(T') = MaxPriorityLevel_{RTOS} + pas$ * sizeOf($Higher-levels(T)$) ;
  sinon
    $PriorityValue(T') = MaxPriorityLevel_{RTOS}$ - pas * sizeOf($Higher-levels(T)$) ;
  Retourner $PriorityValue(T')$;

Algorithme 12 : Algorithme de portage des valeurs de période

Entrées :
$PriorityValue(T)$: la valeur de période d'une tâche T dans le modèle de conception
Résultat :
$PriorityValue(T')$: La valeur de période de la tâche T' qui **correspond à** T au niveau implémentation
Données :
$unit_{conception}$: l'unité de la période de l'instance T au niveau conception
$swPlatform_{RTOS}$: le modèle du RTOS cible
$Clock_{RTOS}$: le temporisateur (*clock*) du RTOS cible
$Tick_{RTOS}$: le tick du temporisateur (*clock*) du RTOS cible
$unit_{RTOS}$: L'unité du *clock tick* au niveau du RTOS cible
$TickValue$: la valeur du *clock tick* du RTOS cible
α : un coefficient **tel que** $unit_{conception} = \alpha unit_{RTOS}$
début
$unit_{conception}$ = getPeriodUnit($PriorityValue(T)$) ;
$Clock_{RTOS}$ = $swPlatform_{RTOS}$.getclock() ;
$Tick_{RTOS}$ = $Clock_{RTOS}$.gettick() ;
$unit_{RTOS}$ = $Tick_{RTOS}$.getTickUnit() ;
$TickValue$ = $Tick_{RTOS}$.getTickValue() ;
$\alpha = \frac{unit_{conception}}{unit_{RTOS}}$;
$PriorityValue(T') = \frac{PriorityValue(T)*\alpha}{TickValue}$;
Retourner $PriorityValue(T')$;

Synthèse

L'algorithme de portage (Algotithme 6) constitue la base de cette phase de portage, qui permet la génération des modèles d'implémentation spécifiques à un RTOS à partir des modèles de conception des applications temps réel. Comme nous l'avons expliqué, cet algorithme considère deux niveaux de portage. Le premier niveau de portage permet d'identifier les concepts du RTOS cible afin d'instancier les éléments du modèle d'implémentation résultant. Tant dit que le deuxième niveau de portage traite le portage des valeurs des propriétés en considérant les caractéristiques du RTOS cible. Nous avons particulièrement détaillé deux exemples de portage des valeurs des propriétés à savoir les priorités et les périodes.

La complexité (au pire cas) de l'algorithme de portage (Algorithme 6) dépend de la complexité des algorithmes de portage des valeurs des propriétés considérés. En effet, cet algorithme fait appel à ces algorithmes de portage ($Portage_P$) dans une boucle de taille n ou n représente le nombre des concepts définis dans un modèle de conception. Ainsi, si la complexité de l'algorithme considéré pour le portage de la valeur de la propriété est constante *O(1)*(comme par exemple l'algorithme de portage des valeurs des périodes), la complexité de cet algorithme est linéaire *O(n)*. Cependant, dans le cas ou la complexité de l'algorithme de portage des valeurs des propriétés est linéaire (comme par exemple l'algorithme de portage

linéaire des des valeurs des priorités que nous avons présenté), la complexité de l'algorithme de portage est polynomiale O(n^2). Ainsi la complexité au pire cas est polynomiale O(n^2), avec n représente le nombre de concepts dans le modèle de conception en entrée. En termes de terminaison, cet algorithme ne présente pas des boucles infinies puisque le nombre de concepts dans le modèle de conception initial est connu à l'avance.

Il est à noter qu'il ne s'agit pas d'une preuve par récurrences pour la terminaison ni d'un calcul détaillé de la complexité. Toutefois, il est important de considérer ces caractéristiques et s'assurer aussi que l'algorithme répond bien au but de l'utilisateur qui est la génération d'un modèle spécifique à un RTOS particulier à partir d'un modèle de conception basé sur une plateforme abstraite d'analyse. Pour cela, considérons un modèle de conception basé sur une configuration d'une plateforme abstraite d'analyse représentée par un extrait de son modèle appelée ici *Analysis_Platform*. De plus, cette figure montre un extrait du modèle du RTOS RTEMS [8] sur lequel le modèle de conception sera déployé. Une condition acquise pour cet algorithme c'est que le nombre des concepts dans le modèle d'implémentation doit être égal au nombre de concepts définis dans le modèle de conception. De plus, seules les propriétés nécessaires pour l'exécution doivent apparaitre dans le modèle résultant.

La Figure 5.3 donne le résultat produit par l'algorithme de portage pour l'exemple précédent 5.2. Notons que pour ce modèle d'implémentation, l'algorithme de portage fait appel à la stratégie linéaire pour le portage des niveaux de priorité. Pour cet exemple, l'algorithme de portage commence par déterminer pour chaque concept du modèle de conception typé par une ressource du modèle de la plateforme d'analyse, le type correspondant dans le modèle de RTEMS en se basant sur les règles de correspondance (C1, C2 et C3) que nous avons identifiées. Le modèle de conception de la Figure 5.2 montre deux concepts qui sont *task1* et *task2* instances de la ressource *PeriodicAnalysis_Task*. Selon les règles de correspondance, *Periodic_RTEMSTask* est la ressource appropriée du modèle de RTEMS qui correspond à la ressource abstraite source *PeriodicAnalysis_Task*. En effet, ces deux ressources sont annotées par le même stéréotype «swTask» et les trois propriétés *priorityElements*, *entryPoints* et *periodElements* référencent des attributs des deux concepts. De plus, la propriété *type* du stéréotype «swTask» a la même valeur qui est égale à *periodic* dans les deux modèles. L'algorithme de portage génère ainsi deux instances typées par *Periodic_RTEMSTask* dans le modèle d'implémentation résultant. En ce qui concerne le deuxième niveau de portage, seules les propriétés *priority*, *functions* et *period* de la classe *PeriodicAnalysis_Task* nécessitent un portage de leurs valeurs (en suivant les règles C4 et C5). Ainsi, l'algorithme de portage fait appel aux algorithmes de portage des valeurs de ces propriétés.

Dans ce qui suit, nous détaillons la phase de validation de portage qui permet de confirmer que les modèles issus de cette phase sont valides.

5.3.2 Phase de validation du Portage

L'objectif de cette phase est de valider la phase de portage précédente. En d'autres termes, elle est en charge de vérifier la conformité du modèle d'implémentation résultant par

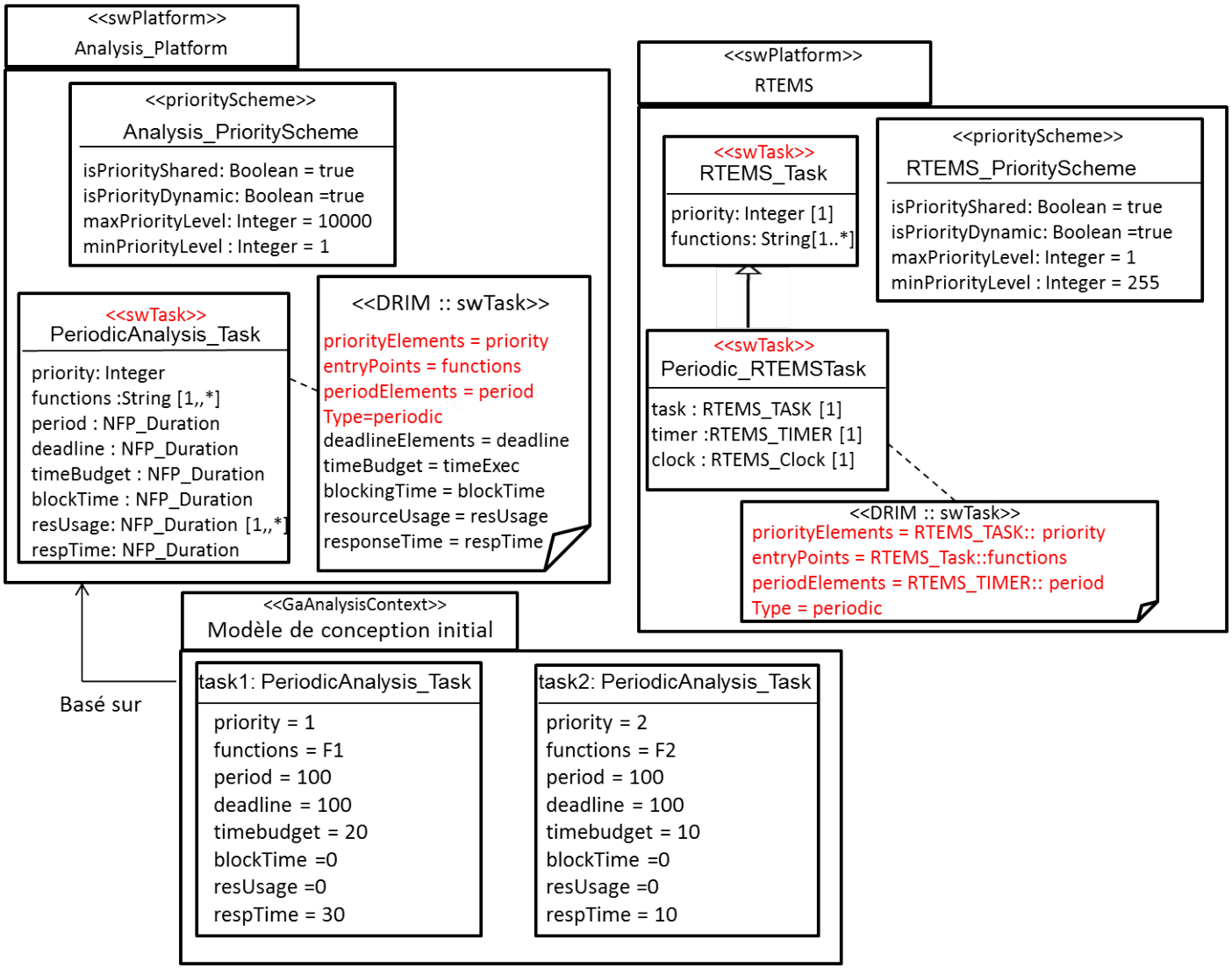


FIGURE 5.2 – *Un exemple de modèle de conception, plateforme abstraite d'analyse et RTOS : entrées pour l'algorithme de portage*

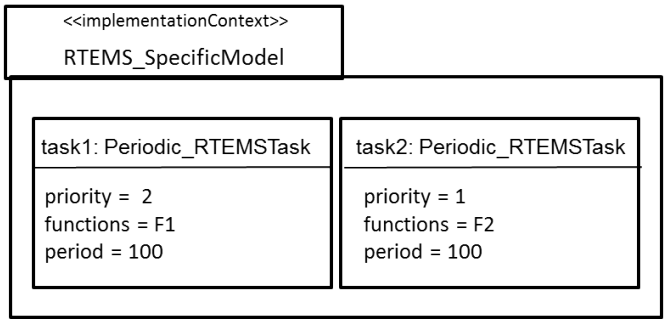


FIGURE 5.3 – *Le modèle spécifique à RTEMS généré par l'algorithme de portage*

rapport au modèle de conception source. Cette vérification sert comme étape préliminaire pour confirmer que les propriétés temporelles qui sont déjà vérifiées au niveau conception restent respectées à l'implémentation. Pour cela, il est nécessaire de définir les propriétés à vérifier sur le modèle d'implémentation obtenu afin de valider la phase de portage. Nous identifions ainsi les trois propriétés P1, P2 et P3 ci-dessous :

- **P1** : les valeurs des niveaux de priorité des différentes tâches utilisées au niveau implémentation, doivent être toujours comprises dans l'intervalle des valeurs de priorité (entre les valeurs maximale et minimale) autorisées par le RTOS cible
- **P2** : l'ordre de précédence entre les niveaux de priorités doit être le même à la conception et à l'implémentation. Par conséquent pour cette propriété nous ne nous intéressons pas aux valeurs de priorités mais plutôt à la relation d'ordre entre les niveaux de priorité qui doit être conservée.
- **P3** : l'ordre d'accès aux ressources partagées doit être conservé. En effet, cette propriété est vérifiée si nous avons la même implémentation de la ressource partagée, en particulier le même protocole de synchronisation (ce qui est assuré par la phase de test de faisabilité). En plus, si l'ordre d'exécution des différentes tâches définies au niveau conception est conservé au niveau implémentation (ce qui est assuré par la vérification de la propriété P2).

L'algorithme 13 donne une description algorithmique du test permettant la vérification de la première propriété sur le modèle d'implémentation généré par la phase de portage.

Cet algorithme vérifie si toutes les valeurs de priorité des tâches du modèle d'implémentation ($Tasks_{implementation}$) produit par la phase de portage sont comprises dans l'intervalle des priorités autorisées par le RTOS cible (i.e. entre les niveaux maximal et minimal de priorité autorisés par le RTOS). Si ce n'est pas le cas, cet algorithme génère une erreur.

La complexité de cet algorithme est linéaire *(O(n))* avec *n* représente le nombre de tâches dans le modèle d'implémentation généré. Cet algorithme ne présente pas de boucles infinies puisque le nombre de répétions des boucles utilisées dans cet algorithme est connu à l'avance.

La vérification de cette propriété *P1* sur le modèle d'implémentation généré par la phase de portage et donné par la Figure 5.3 précédente, produit un retour positif (aucune er-

reur). En effet, ce modèle spécifique à RTEMS montre deux tâches appelées respectivement *task1* et *task2* instances de la ressource *Periodic_RTEMSTask*. La valeur de la priorité de la tâche *task1* est égale à 2 et celle de la tâche *task2* est égale à 1. Nous pouvons remarquer que ces deux valeurs sont bien comprises dans l'intervalle des niveaux de priorité autorisés par RTEMS qui est [1,255] comme le montre son modèle présenté dans la Figure 5.2.

Algorithme 13 : Algorithme de test de la propriété P1

Entrées :
$Tasks_{implementation}$: une liste des tâches définies dans le modèle d'implémentation
$swPlatform_{RTOS}$: le modèle du RTOS cible
Résultat :
Verdict S= {E, NP} ; E : Erreur, NP : Pas de Problème
Données :
$priorityScheme_{RTOS}$:le schéma de priorité du RTOS cible
$MaxPriorityLevel_{RTOS}$:le niveau maximal de priorité dans le RTOS cible
$MinPriorityLevel_{RTOS}$:le niveau minimal de priorité dans le RTOS cible
$PriorityValue(T)$: la valeur de priorité de la tâche T
Initialisations :
$S \leftarrow NP$;
début
    $priorityScheme_{RTOS}$= $swPlatform_{RTOS}$.getpriorityScheme() ;
    $MaxPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getmaxPriorityLevel() ;
    $MinPriorityLevel_{RTOS}$ = $swPlatform_{RTOS}$.getminpriorityLevel() ;
    si $MaxPriorityLevel_{RTOS} < MinPriorityLevel_{RTOS}$ **alors**
        pour tout $T \in Tasks_{implementation}$ **faire**
            $PriorityValue(T)$ = T.getpriorityValue() ;
            si $PriorityValue(T) < MaxPriorityLevel_{RTOS}$ **ou** $PriorityValue(T) > MinPriorityLevel_{RTOS}$ **alors**
                $S \leftarrow E$;
    sinon
        pour tout $T \in Tasks_{implementation}$ **faire**
            $PriorityValue(T)$ = T.getpriorityValue() ;
            si $PriorityValue(T) > MaxPriorityLevel_{RTOS}$ **ou** $PriorityValue(T) < MinPriorityLevel_{RTOS}$ **alors**
                $S \leftarrow E$;
    Retourner S ;

La vérification de la deuxième propriété (P2), nécessite en plus la définition du méta-modèle présenté dans la Figure 5.4. Dans ce méta-modèle, l'application temps réel est traduite par une classe (*classe Application*). Cette classe définit un attribut *Name* qui indique le nom de *l'application*. Puisque nous nous intéressons à l'ordre d'exécution des différentes tâches pour vérifier la propriété P2, nous spécifions une classe nommée *PriorityLevel* représentant les niveaux de priorité distincts définis dans l'application. Ces niveaux peuvent également avoir une relation de précédence. Cette relation de précédence est traduite par l'as-

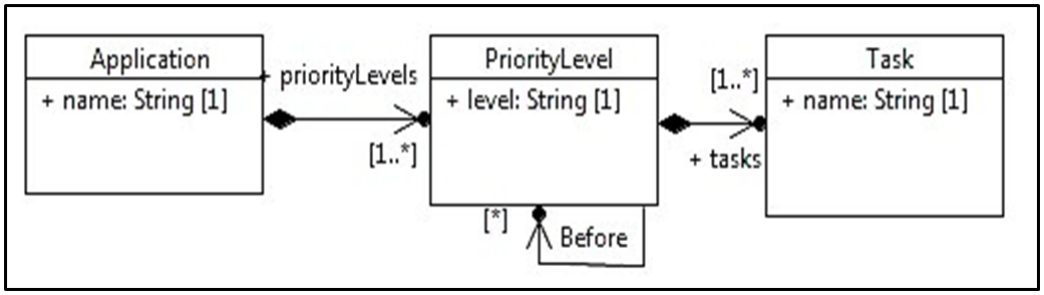


FIGURE 5.4 – *Le méta-modèle utilisé pour la vérification de la propriété P2*

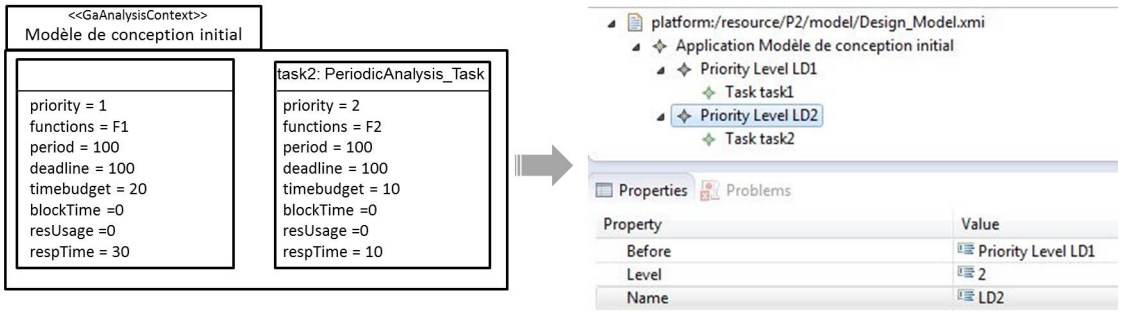


FIGURE 5.5 – *Modèle de conception de la Figure 5.2 comme instance du méta-modèle utilisé pour la vérification de la propriété P2*

sociation *Before*. Par conséquent, ce méta-modèle considère que le plus important n'est pas les valeurs de priorité, mais la relation de précédence entre eux. Chaque niveau de priorité peut contenir une ou plusieurs tâches *Task* qui partagent ce niveau.

Ainsi, afin de vérifier la propriété P2, les modèles de conception et d'implémentation sont transformés en des modèles conformes à ce méta-modèle. La vérification de P2 revient donc à vérifier si les modèles obtenus de cette transformation sont équivalents (i.e. même si les valeurs des priorités tâches sont différentes, l'ordre de précédence des tâches est conservé). Si c'est le cas, P2 est alors vérifiée.

Afin de vérifier cette propriété *P2* pour l'exemple précédent illustré par les Figures 5.2 et 5.3, il faut transformer les modèles de conception et d'implémentation en des modèles conformes au méta-modèle de la Figure 5.4.

La Figure 5.5 montre l'application nommée *Modèle de conception initial* constituée de deux niveaux de priorité appelés respectivement *LD1* et *LD2*. Le niveau *LD1* correspond au niveau de priorité *1* du modèle de conception et contient la tâche *task1*.Tant dit que le niveau *LD2* correspond au niveau de priorité 2 du modèle de conception et contient la tâche *task2*. Selon la configuration utilisée de la plateforme abstraite d'analyse et dont le modèle est présenté dans la Figure 5.2, l'ordre de priorité est croissant (i.e. le niveau *2* est plus prioritaire que le niveau *1*). Par conséquent, comme illustré dans la Figure 5.5, le niveau *LD2*(contenant la tâche *task2*) précède (*Before*) le niveau *LD1* (contenant la tâche *task1*).

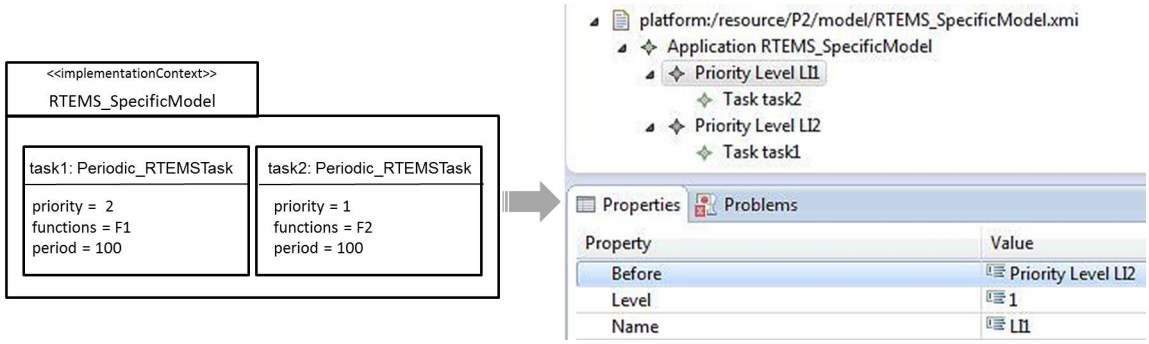


FIGURE 5.6 – *Modèle d'implémentation de la Figure 5.3 comme instance du méta-modèle utilisé pour la vérification de la propriété P2*

De même, le modèle d'implémentation spécifique à RTEMS donné par la Figure 5.3 est transformé en un modèle conforme au méta-modèle proposé pour la vérification de la propriété *P2*. En effet, la Figure 5.6 montre l'application nommée *RTEMS_SpecificModel* constituée de deux niveaux de priorité appelés respectivement *LI1* et *LI2*. Le niveau *LI1* correspond au niveau de priorité *1* du modèle d'implémentation et contient la tâche *task2*. Tant dit que le niveau *LI2* correspond au niveau de priorité *2* du modèle de conception et contient la tâche *task1*. Selon le modèle de RTEMS présenté dans la Figure 5.2, l'ordre de priorité est décroissant (i.e. le niveau *1* est plus prioritaire que le niveau *2*). Par conséquent, comme illustré dans la Figure 5.6, le niveau *LI1*(contenant la tâche *task2*) précède (*Before*) le niveau *LI2* (contenant la tâche *task1*).

Nous pouvons ainsi conclure que même si les valeurs des priorités des tâches qui décrivent une même application à deux niveaux différents (au niveau conception et implémentation) sont différentes, l'ordre de précédence entre ces deux tâches est conservé (toujours *task2* précéde *task1*). P2 est alors vérifiée.

5.4 Phase de refactoring

Dans le cas où la phase d'évaluation de faisabilité détecte un problème de déploiement, l'objectif de cette phase est de trouver une solution implémentable pour le modèle de conception initial. Ceci revient à appliquer le patron approprié à partir d'une base de patrons prédéfinis. Dans ce chapitre, nous détaillons quatre exemples de patrons que nous avons proposés dans le but de répondre à des problèmes de déploiement. Ainsi, pour chaque patron nous expliquons le problème de déploiement qu'il traite (à partir de la liste des tests de la phase d'évaluation de faisabilité de déploiement) et nous décrivons la solution considérée. Le contexte des différents patrons est par défaut le modèle de conception initial (qui est le point d'entrée de la phase de refactoring). Cependant, lorsque des hypothèses supplémentaires doivent être considérées, elles sont mentionnées au moment de la description de la solution du patron.

5.4.1 NPAP : Un patron pour une nouvelle assignation des valeurs des priorités

Dans cette section, nous expliquons le problème de déploiement lié à ce patron et la solution proposée. Pour cela, nous désignons par N le nombre des niveaux de priorité distincts autorisé par le RTOS cible pour l'application considérée et par n le nombre des niveaux de priorité distincts utilisé au niveau conception pour décrire l'application.

Énoncé du problème Une fois que la phase d'évaluation de faisabilité détecte que le nombre des niveaux de priorité distincts utilisé au niveau conception pour décrire l'application est supérieur au nombre autorisé par le RTOS (c'est-à-dire $n \geq N$), ce patron peut être appliqué. En d'autres termes, ce patron est invoqué au cas où le test des niveaux de priorité distincts (Algorithme3) produit une erreur.

Description de la solution La solution qui décrit ce patron est basée sur les résultats des travaux présentés dans [53]. En effet, ce travail discute différents problèmes liés à l'analyse d'ordonnançabilité temps réel. En particulier, il invoque le problème de dégradation des performances en termes d'ordonnançabilité au cas où le nombre de niveaux de priorité distincts autorisé par une plateforme est insuffisant pour implémenter une application temps réel. Ainsi, ce travail propose une solution basée sur une nouvelle assignation des niveaux de priorité en fonction du nombre autorisé par la plateforme cible.

La solution proposée n'est applicable que si toutes les tâches qui décrivent l'application sont *indépendantes*. Par ailleurs, *les échéances de toutes les tâches doivent être égales à leurs périodes* (c'est-à-dire $\forall T_i \in M_{conception}, D_i = P_i$). Ainsi, ce patron n'est applicable que si le modèle de conception initial satisfait ces hypothèses.

Pour chaque groupe de tâches ayant des périodes appartenant à un intervalle donné, ce patron attribue un niveau de priorité unique à toutes ces tâches. Les limites des différents intervalles sont déterminées en fonction d'un paramètre r que nous définissons comme suit :

$$r = \frac{P_{max}}{P_{min}}^{\frac{1}{N}} \tag{5.1}$$

Dans cette expression, P_{min} et P_{max} représentent respectivement la période minimale et maximale de toutes les tâches définies dans le modèle de conception. N représente le nombre des niveaux de priorité distincts autorisé par le RTOS pour l'application considérée. Par conséquent, les limites des intervalles sont calculées comme suit : P_{min},rP_{min},r^2P_{min},..., P_{max}. Ainsi, toutes les tâches ayant des périodes entre P_{min} et rP_{min} auront le niveau de priorité le plus haut. Les tâches ayant des périodes entre rP_{min} et r^2P_{min} auront le niveau de priorité suivant, etc. Une description algorithmique de ce patron est donnée par l'algorithme 14.

Algorithme 14 : Algorithme du patron NPAP

Entrées :

$M_{conception}$: le modèle de conception initial de l'application

$swPlatform_{analysis}$: le modèle de la plateforme abstraite d'analyse (i.e. une configuration)

N : le nombre de niveaux de priorité distinct autorisé par le RTOS pour l'application considérée

Résultat :

$M_{modifie}$: le modèle de conception modifié suite à l'application du patron

Données :

$MaxPeriodValue$: la période la plus grande dans le modèle de conception initial $M_{conception}$

$MinPeriodValue$: la période la plus petite dans le modèle de conception initial $M_{conception}$

$priorityScheme_{analysis}$:le schéma de priorité du RTOS cible

$MaxPriorityLevel_{analysis}$:le niveau maximal de priorité dans le RTOS cible

$MinPriorityLevel_{analysis}$:le niveau minimal de priorité dans le RTOS cible

$Tasks$: la liste des tâches dans le modèle de conception modifié

$PriorityValue(T)$: la valeur de priorité de la tâche T

$PeriodValue(T)$: la valeur de période de la tâche T

r : le paramètre donné par l'équation 5.1

Initialisations :

Définition d'un compteur $Counter \leftarrow 0$;

Définition d'une variable intermédiaire $e \leftarrow 1$;

$M_{modifie} \leftarrow M_{conception}$;

début

- $priorityScheme_{analysis}$= $swPlatform_{analysis}$.getpriorityScheme() ;
- $MaxPriorityLevel_{analysis}$ = $swPlatform_{analysis}$.getmaxPriorityLevel() ;
- $MinPriorityLevel_{analysis}$ = $swPlatform_{analysis}$.getminpriorityLevel() ;
- $r = \frac{MaxPeriodValue}{MinPeriodValue}^{\frac{1}{N}}$;
- **si** $MaxPriorityLevel_{analysis}$ ==*0* **alors**
  - **tant que** *isDifferent(*$MaxPriorityLevel_{analysis}$ + *Counter, N-1)* **faire**
    - **pour tout** $i \in Tasks$ **faire**
      - **si** $PeriodValue(i) \in [e * MinPeriodValue, r * MinPeriodValue]$ **alors**
        - $PriorityValue(T) = MaxPriorityLevel_{analysis} + Counter$;
        - $e = r$; $r = r * r$;$Counter = Counter$+1 ;
- **si** $MaxPriorityLevel_{analysis} > MinPriorityLevel_{analysis}$ **alors**
  - **tant que** *isDifferent(*$MaxPriorityLevel_{analysis}$- *Counter,* $MaxPriorityLevel_{analysis}$- *N)* **faire**
    - **pour tout** $i \in Tasks$ **faire**
      - **si** $PeriodValue(i) \in [e * MinPeriodValue, r * MinPeriodValue]$ **alors**
        - $PriorityValue(T) = MaxPriorityLevel_{analysis}$ - $Counter$;
        - $e = r$; $r = r * r$;$Counter = Counter$+1 ;
- **sinon**
  - **tant que** *isDifferent(*$MaxPriorityLevel_{analysis}$ + *Counter, N)* **faire**
    - **pour tout** $i \in Tasks$ **faire**
      - **si** $PeriodValue(i) \in [e * MinPeriodValue, r * MinPeriodValue]$ **alors**
        - $PriorityValue(T) = MaxPriorityLevel_{analysis} + Counter$;
        - $e = r$; $r = r * r$;$Counter = Counter$+1 ;
- **Retourner** $M_{modifie}$;

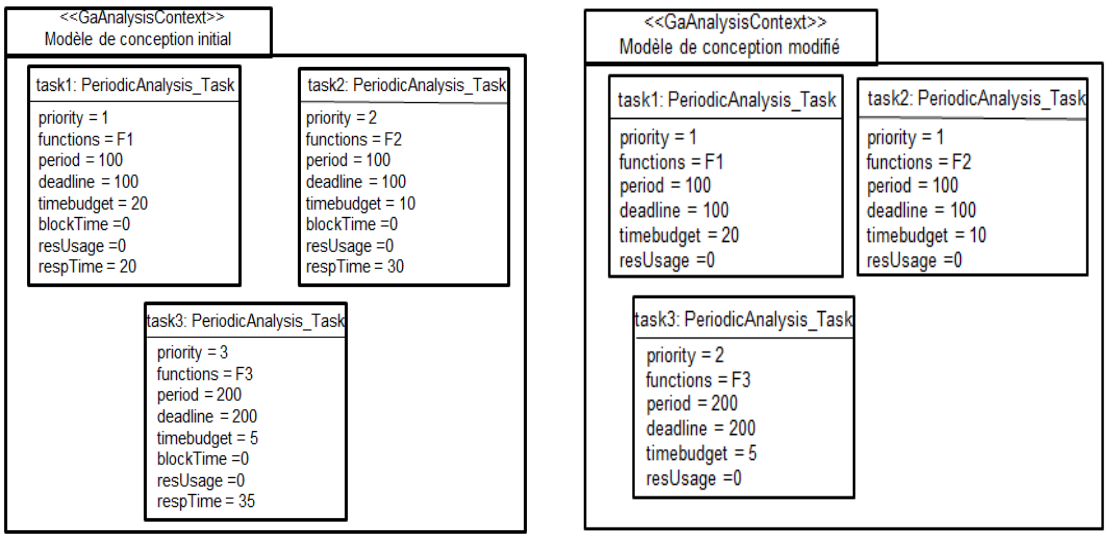
Cet algorithme définit comme entrées le modèle de conception de l'application ($M_{conception}$), le modèle de la plateforme abstraite d'analyse ($swPlatform_{analysis}$) et le nombre des niveaux de priorité distincts autorisé par le RTOS pour implémenter $M_{conception}$. S'il est applicable (i.e. le modèle de conception initial satisfait les hypothèses que nous avons spécifiées pour ce patron), cet algorithme produit comme sortie le modèle de conception modifié. Pour cela, cet algorithme commence par calculer la valeur du paramètre r sur lequel se base ce patron en utilisant la période maximale et la période minimale des tâches du modèle d'une part et le nombre N d'autre part. Les priorités des tâches seront ainsi assignées selon leur période et la valeur du paramètre r. Le tableau 5.4 donne les caractéristiques de l'algorithme 14 en termes de complexité (au pire cas) et de terminaison.

TABLE 5.2 – Les caractéristiques de l'algorithme NPAP en termes de complexité et de terminaison

	Calcul de la complexité	**Absence des boucles infinies**
Algorithme du patron NPAP	La complexité de cet algorithme est linéaire ($O(n)$) avec n représente le nombre de tâches du modèle de conception	Cet algorithme se termine quand le nombre des niveaux de priorité distincts utilisés dans le modèle de conception sera égal à N.

Afin de montrer que l'algorithme 14 répond correctement au besoin de l'utilisateur qui est la réduction du nombre des niveaux de priorité distincts utilisé au niveau conception dans le but de s'adapter aux contraintes d'implémentation, nous considérons l'exemple de modèle de conception initial illustré par la Figure 5.7(a).Nous supposons que ce modèle repose sur une configuration d'une plateforme abstraite d'analyse qui définit un ordre *décroissant* des niveaux de priorité (1 est le niveau le plus prioritaire) et que le RTOS cible (sur lequel ce modèle sera déployé) n'autorise que *deux niveaux* de priorité pour l'implémentation de ce modèle. Une condition requise pour cet algorithme c'est que le nombre de tâches suite à l'application du patron est conservé. Ainsi, seules les valeurs des priorités des tâches changent dans le but de diminuer le nombre des niveaux de priorité distincts à 2.

Le modèle de conception initial satisfait les hypothèses du patron NPAP (i.e.toutes les tâches sont indépendantes ayant des périodes égales à leurs échéances)et présente trois niveaux de priorités distincts. Ainsi, le résultat d'application du patron NPAP dans le but de réduire le nombre des niveaux de priorité distincts de trois à deux niveaux est donné par la Figure 5.7(b). Afin de générer ce modèle, l'algorithme 14 calcule la valeur de r qui est égale à $\sqrt{2}$ pour cet exemple ($\frac{200}{100}^{\frac{1}{2}} = \sqrt{2} = 1.414$), puis procède à une nouvelle affectation des valeurs de priorité aux différentes tâches. Il donne ainsi le niveau le plus haut, qui est égal à 1 pour cet exemple, aux tâches ayant des périodes comprises dans l'intervalle [100,141.42]($[P_{min}, r * P_{min}]$) qui sont *task1* et *task2*. Ensuite, il affecte le niveau qui suit (i.e. le niveau 2) à la tâche *task3* dont la valeur de période est comprise entre [141.42,200] ($[r * P_{min}, r^2 * P_{min}]$. Il est clair alors que, comme le modèle de conception initial, le modèle de conception modifié consiste aussi de trois tâches mais qui sont définies avec deux niveaux de



(a) Modèle de conception initial (b) Modèle de conception modifié

FIGURE 5.7 – Exemple d'application du patron NPAP

priorité distincts. Suite à l'application de ce patron et la génération du modèle de conception modifié, la phase de refactoring procède à l'analyse de ce modèle (i.e. calcul du temps de blocage et le temps de réponse de chaque tâche) afin de vérifier si les contraintes de temps de l'application sont toujours satisfaites.

5.4.2 DPMP : Un patron pour la fusion des tâches de priorités distinctes

Le problème de déploiement lié à ce patron est le même problème que le patron précédent (*NPAP*). Cependant, ce patron permet de considérer des modèles de conception plus génériques que le patron précédent en permettant le partage des ressources entre les tâches du modèles (tâches dépendantes) et une relation aléatoire entre l'échéance d'une tâche et sa période. Dans cette section, nous rappelons le problème de déploiement lié à ce patron et nous expliquons la solution proposée.

Énoncé du problème Une fois que la phase d'évaluation de faisabilité détecte que le nombre des niveaux de priorité distincts utilisé au niveau conception pour décrire l'application est supérieur au nombre autorisé par le RTOS (c'est-à-dire $n \geq N$), ce patron peut être appliqué. En d'autres termes, ce patron est invoqué au cas où le test des niveaux de priorité distincts (Algorithme3) produit une erreur.

Description de la solution Considérons un modèle de conception initial que nous désignons par $M_{conception} = \{T_1, T_2, \ldots, T_m\}$. Ce modèle définit m tâches et n niveaux de priorité distincts ($n \geq m$). Nous considérons aussi que chaque tâche $T_i \in M_{conception}$ est caractérisée par un ensemble de paramètres : $T_i = (p_i, C_i, P_i, D_i, f_i, R_i, U_{R_i,T_i})$ avec :

- p_i :la priorité de la tâche T_i
- C_i : le temps d'exécution au pire cas (*wcet*) de la tâche T_i
- P_i : le taux d'activation de la tâche T_i
- D_i : l'échéance de la tâche T_i
- f_i : l'ensemble des fonctions exécutées par la tâche T_i
- R_i : l'ensemble des ressources partagées utilisées par la tâche T_i
- U_{R_i,T_i} : l'utilisation de la R_i (ou de l'ensemble des ressources R_i) par la tâche T_i

Afin de réduire le nombre n pour qu'il soit égal à N, ce patron fusionne les tâches ayant, d'une part, des niveaux de priorité distincts et d'autre part des *taux harmoniques* (i.e. deux tâches T_i et T_j sont dites harmoniques *si et seulement si* $P_i mod P_j = 0$ et $P_j \geq P_i$). Cette deuxième condition (les taux harmoniques) permet de préserver la spécification de haut niveau en termes de taux d'exécution des fonctions applicatives.

Ainsi, la fusion de deux tâches $T_i = (p_i, C_i, P_i, D_i, f_i, R_i, U_{R_i,T_i})$ et $T_j = (p_j, C_j, P_j, D_j, f_j, R_j, U_{R_j,T_j})$ tels que $p_i \neq p_j$ et $\frac{P_j}{P_i} = q$ avec q est un entier strictement positif est donnée par la figure suivante :

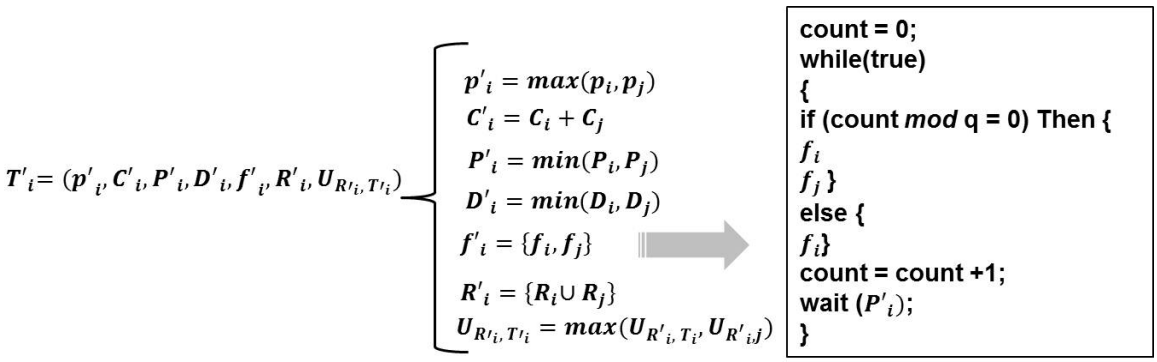


FIGURE 5.8 – *Fusion de deux tâches en suivant le patron DPMP*

La Figure 5.8 montre les paramètres de la tâche T'_i qui résulte de la fusion des deux tâches T_i et T_j. En effet, suite à cette fusion le modèle résultant consiste de *m-1* tâches et *n-1* niveaux de priorité distincts. Notons aussi que ce patron permet de fusionner plus que deux tâches à la fois et que cette opération de fusion peut se répéter plusieurs fois jusqu'à ce que n soit égale N.

Ainsi, pour un modèle de conception initial (n niveaux distincts), ce patron doit trouver une façon pour fusionner les tâches du modèle initial afin de réduire le nombre n pour qu'il soit égal à N. Une description algorithmique de ce patron s'avère non suffisante. En effet, pour un problème donné, plusieurs solutions (différentes manières de fusionner les tâches) sont possibles. Ce problème est dit *combinatoire*. La Figure 5.9 montre les différentes solutions possibles pour un modèle de conception initial constitué de quatre tâches et pour un nombre N égal à deux. Cette figure montre que cinq solutions sont possibles d'une part et que la façon de fusionner les tâches a un impact majeur sur les performances du modèle résultant. En effet, pour cet exemple, l'estimation des performances en termes d'utilisation processeur

montre que les solutions M_1 et M_2 sont les meilleures.

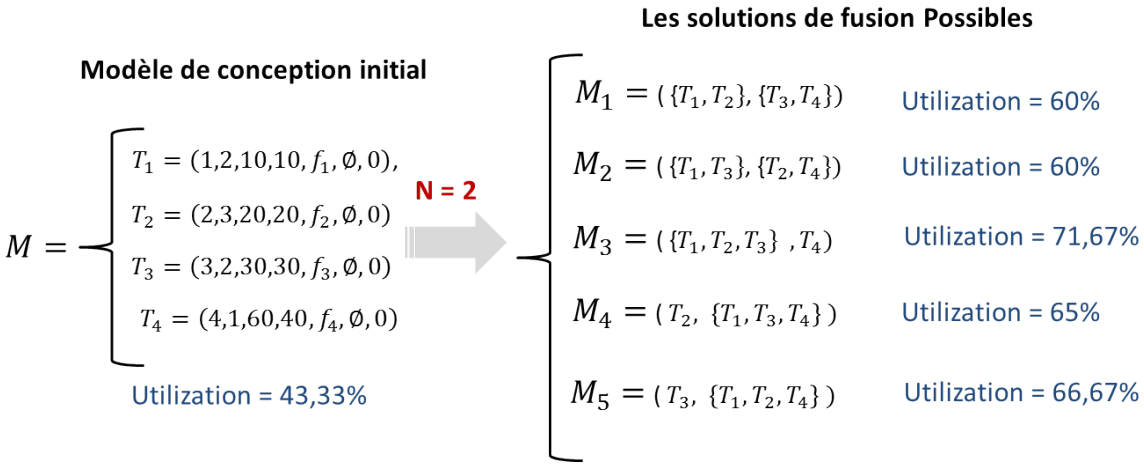


FIGURE 5.9 – *DPMP :Un problème combinatoire*

A partir de toutes ces considérations, nous proposons de formuler le patron DPMP en utilisant les techniques MILP (Mixed Integer Linear Programming)[44]. Cette formulation est donnée en détails dans l'annexe A. La formulation de ce patron en utilisant ces techniques, permet de confirmer (avant d'appliquer le patron) s'il existe une façon de fusionner les tâches pour un problème donné d'une part et de trouver la meilleure façon (c'est-à-dire la plus optimisée en termes de performances) d'autre part.

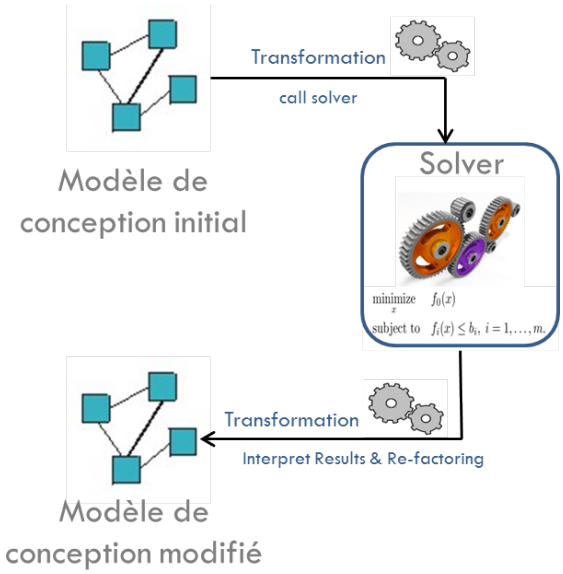


FIGURE 5.10 – *Application du patron DPMP*

Ainsi, comme illustré dans la Figure 5.10, à partir d'un modèle de conception initial, nous définissons une transformation qui permet d'appeler le solveur et d'exécuter le programme

linéaire qui implémente ce patron (un extrait de cette transformation est donné aussi en annexe A). Le résultat produit par le solveur est interprété afin de pouvoir générer le modèle de conception modifié.

5.4.3 EPMP : Un patron pour la fusion des tâches de priorités égales

Dans cette section, nous expliquons le problème lié à au patron EPMP et nous décrivons la solution proposée.

Énoncé du problème Au cas où le modèle de conception initial définit au moins deux tâches qui partagent un même niveau de priorité, d'une part, et le RTOS cible n'autorise pas une telle situation d'autre part, ce patron peut être appliqué. En d'autres termes, ce patron est invoqué au cas où le test des niveaux de priorité égaux (Algorithme 4) produit une erreur.

Description de la solution La solution décrivant ce patron est très similaire à celle du patron précédent (DPMP). Cependant, contrairement au patron précédent, ce patron fusionne les tâches de priorités égales afin d'éviter le partage d'un même niveau entre les tâches. Ainsi, ce patron fusionne les tâches ayant, d'une part, des niveaux de priorité égaux et d'autre part des taux harmoniques.

Considérons un modèle de conception initial que nous désignons par $M_{conception} = \{T_1, T_2, \ldots, T_m\}$. Ce modèle définit m tâches et n niveaux de priorité distincts ($n \geq m$).Lorsque $n < m$, cela signifie qu'il existe au moins deux tâches $T_i = (p_i, C_i, P_i, D_i, B_i, f_i, R_i)$ et $T_j = (p_j, C_j, P_j, D_j, B_j, f_j, R_j)$ appartenant à $M_{conception}$ tels que $p_i = p_j$. Si en plus, ces deux tâches possèdent des taux harmoniques $\frac{P_j}{P_i} = q$ avec q est un entier strictement positif et $P_j \geq P_i$, ce patron fusionne ces deux tâches en une seule que nous désignons par T'_i comme suit :

$$T'_i = (p'_i, C'_i, P'_i, D'_i, f'_i, R'_i, U_{R'_i, T'_i})$$

$$\begin{cases} p'_i = p_i = p_j \\ C'_i = C_i + C_j \\ P'_i = min(P_i, P_j) \\ D'_i = min(D_i, D_j) \\ f'_i = \{f_i, f_j\} \\ R'_i = \{R_i \cup R_j\} \\ U_{R'_i, T'_i} = max(U_{R'_i, T_i}, U_{R'_i j}) \end{cases}$$

```
count = 0;
while(true)
{
if (count mod q = 0) Then {
f_i
f_j }
else {
f_i}
count = count +1;
wait (P'_i);
}
```

FIGURE 5.11 – *Fusion de deux tâches en suivant le patron EMPM*

Suite à cette fusion le modèle résultant consiste de *m-1* tâches et *n* niveaux de priorité distincts. Notons aussi que si le modèle de conception initial présente plus que deux tâches partageant un même niveau, toutes ces tâches seront fusionnées en une seule tâche. Ainsi,

pour un problème donné, une solution unique pour ce patron consiste à fusionner en une seule tâche toutes les tâches appartenant à un même niveau. Une description algorithmique de ce patron est donnée par l'algorithme 15.

L'algorithme 15 définit comme entrée le modèle de conception initial de l'application ($M_{conception}$) et produit comme sortie le modèle de conception modifié ($M_{modifie}$). L'idée de cet algorithme est crée une liste (E_{Tasks}) pour chaque niveau de priorité défini dans le modèle de conception initial. Dans le cas ou la taille de cette liste est supérieur à 1, ceci veut dire qu'il existe plus qu'une tâche qui partage ce niveau. Ainsi, toutes les tâches appartenant à ce niveau seront fusionnées.

Algorithme 15 : Algorithme du patron EPMP

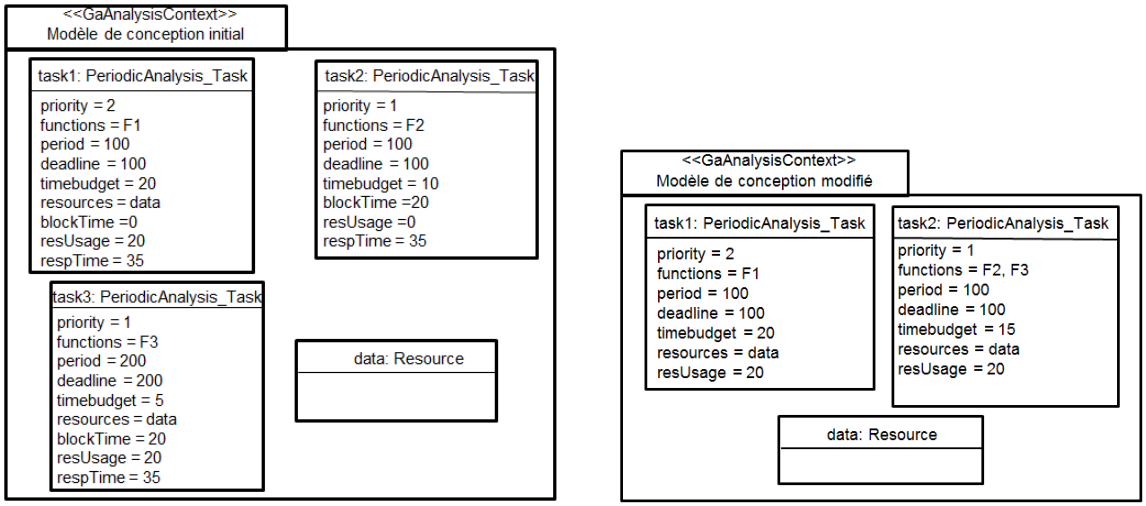
Entrées :
$M_{conception}$: le modèle de conception initial de l'application
Résultat :
$M_{modifie}$: le modèle de conception modifié suite à l'application du patron
Données :
$Tasks$: la liste des tâches dans le modèle de conception initial
$Ref_{Priority}$: le niveau de priorité référence
Ref_{Task} : la tâche référence ayant une valeur de priorité égale à $Ref_{Priority}$
$Current_{priority}$: la priorité qui référence le niveau actuel
E_{Tasks} : la liste des tâches ayant des niveaux de priorité égaux à $Current_{priority}$
début
    $Ref_{Priority}$ = getHighestPriority($M_{conception}$) ;
    $Current_{priority}$ = $Ref_{Priority}$;
    Ref_{Task} = getReferenceTask($M_{conception}$, $Current_{priority}$) ;
    tant que Ref_{Task} != **null faire**
        pour tout $i \in Tasks$ **faire**
            si *isEqualPriority(i,*Ref_{Task}*)* **alors**
                si *isHarmonicPeriod(i,*Ref_{Task}*)* **alors**
                    ajouter (i,E_{Tasks}) ;
        si *sizeOf(*E_{Tasks}*) >1* **alors**
            $M_{modifie}$ = Merge (E_{Tasks}) ;
            $Current_{priority}$ = getHighestPriority($M_{conception}$, $Ref_{Priority}$) ;
            $Ref_{Priority}$ = $Current_{priority}$;
            Ref_{Task} = getReferenceTask($M_{conception}$, $Current_{priority}$) ;
    Retourner $M_{modifie}$;

Afin de montrer que l'algorithme 15 répond correctement au besoin de l'utilisateur qui est la fusion des tâches définies avec le même niveau de priorité dans le modèle de conception initial, nous considérons l'exemple du modèle de conception initial illustré par la Figure 5.12(a). Nous supposons que ce modèle repose sur une configuration d'une plateforme abstraite d'analyse qui définit un ordre *décroissant* des niveaux de priorité (1 est le niveau le plus prioritaire) et qu'une implémentation basée sur le protocole de synchronisation *PIP* est

TABLE 5.3 – Les caractéristiques de l'algorithme EPMP en termes de complexité et de terminaison

	Calcul de la complexité	**Absence des boucles infinies**
Algorithme du patron EPMP	Cet algorithme contient deux boucles imbriquées de taille n avec n représente le nombre de tâches dans le modèle de conception initial. Par conséquent, la complexité de cet algorithme est polynomiale ($O(n^2)$)	Cet algorithme se termine quand toutes les tâches du modèle de conception initial ayant toutes des niveaux de priorité distincts ont été traitées (Ref_{Task} égal à *null*). Ainsi, cet algorithme ne contient pas des boucles infinies puisque le nombre de tâches dans le modèle de conception initial est borné.

considérée pour les ressources partagées. D'autre part, nous supposons que le RTOS cible (sur lequel ce modèle sera déployé) n'autorise pas le partage d'un même niveau de priorité entre tâches. Une condition requise pour cet algorithme c'est que toutes les tâches dans le modèle de conception résultant de l'application de patron auront des niveaux de priorité distincts.



(a) Modèle de conception initial (b) Modèle de conception modifié

FIGURE 5.12 – Exemple d'application du patron EPMP

Le modèle de conception initial montre deux tâches *task2* et *task3* qui partagent le même niveau de priorité qui est égal 1. Ainsi, le résultat d'application du patron EPMP dans le but d'éliminer cette situation (supposée non supportée par le RTOS cible sur lequel cette application sera déployée) est donnée par la Figure 5.12(b). Dans ce modèle les deux tâches *task2* et *task3* sont fusionnées en une seule tâche *task2*. Les paramètres de la tâche résultante sont

déterminés selon la solution présentée 5.11. Ainsi, le modèle de conception modifié résultant consiste de tâches ayant toutes des niveaux de priorité distincts. Suite à l'application de ce patron et la génération du modèle de conception modifié, la phase de refactoring procède à l'analyse de ce modèle (i.e. calcul du temps de blocage et le temps de réponse de chaque tâche) afin de vérifier si les contraintes de temps de l'application sont toujours satisfaites.

5.4.4 SRMP : Un patron pour la fusion des ressources partagées

Les choix d'implémentation des ressources partagées entre les tâches dans le modèle de conception, ont un impact direct sur les résultats d'analyse, en particulier le protocole de synchronisation. Le test détaillé dans l'algorithme 2 produit une erreur dans le cas où le RTOS cible ne permet pas l'implémentation de la ressource partagée avec le même protocole de synchronisation utilisé au niveau conception pour vérifier les contraintes de temps. Ainsi, dans ce cas, une solution possible qui peut être proposée par la phase de refactoring est de changer ces choix selon ce qui est fourni par le RTOS. Ensuite, revalider le modèle avec les nouveaux choix pour vérifier si les contraintes de temps sont toujours respectées.

Cependant, dans certains cas, une simple modification du protocole de synchronisation peut s'avérer non suffisante à cause des problèmes d'inversion de priorité et d'inter-blocage qui peuvent apparaitre. C'est dans ce contexte que ce patron de fusion des ressources partagées a été proposé. Dans cette section, nous expliquons le problème lié à ce patron et nous décrivons la solution proposée.

Énoncé du problème Au cas où le modèle de conception initial est constitué de tâches qui partagent plus qu'une ressource à la fois, implémentées en utilisant le protocole de synchronisation *PCP* d'une part et que d'autre part le RTOS cible ne fournit quant à lui que *PIP* comme protocole de synchronisation. Dans ce cas, remplacer *PCP* par *PIP* au niveau conception et revalider semble insuffisant à cause du problème d'inter-blocage (expliqué plus bas) qui peut apparaitre à l'implémentation. Ainsi, ce patron doit être appliqué avant d'utiliser *PIP* comme protocole de synchronisation.

Description de la solution En effet, un problème d'inter-blocage peut apparaitre au cas où il existe au moins deux tâches parmi les tâches qui décrivent l'application qui partagent plus qu'une ressource à la fois. Ainsi, la solution proposée par ce patron consiste à fusionner les ressources partagées en une seule ressource afin d'éviter le risque d'inter-blocage en utilisant *PIP*.

Pour cela, considérons un modèle de conception initial composé de m tâches $M_{conception} = \{T_1, T_2, \ldots, T_m\}$ et p ressources $\{R_1, R_2, \ldots., R_p\}$ qui peuvent être partagées entre ces tâches. Supposons que les deux tâches T_i et T_j appartenant à $M_{conception}$ partagent à la fois les deux ressources R_n et $R_m \in \{R_1, R_2, \ldots., R_p\}$. Nous désignons par $U_{(R_i,T_j)}$ l'utilisation de la ressource R_i par la tâche T_j. Dans ce cas, à l'implémentation trois situations sont possibles pour

T_i (respectivement pour T_j) pour accéder aux deux ressources R_n et R_m. La figure 5.13 montre ces différentes situations.

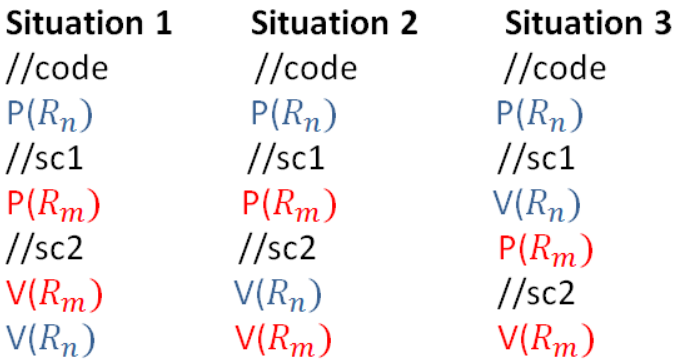


FIGURE 5.13 – *Les différentes situations possibles à l'implémentation pour une tâche T_i afin d'accéder aux deux ressources R_n et R_m*

Au niveau implémentation, une situation d'inter-blocage peut se produire lorsque T_i et T_j accèdent aux ressources R_n et R_m d'une façon imbriquée. Par conséquent, les situations 1 et 2 de la Figure 5.13 correspondent à des situations potentielles d'inter-blocage. Dans la première situation, l'accès et la libération de la ressource R_m se fait durant la section critique de la ressource R_n. Toutefois, dans la deuxième situation, l'accès à la ressource R_m se fait durant la section de la ressource R_n et la libération de cette ressource se fait après la libération de la ressource R_n.

Comme au niveau conception, nous n'avons pas d'idée sur la façon avec laquelle l'accès aux ressources est réalisé, ce patron propose de fusionner les deux ressources R_n et R_m en une seule ressource R'_n ($R'_n = R_n \cup R_m$) dans le but d'éviter le risque d'inter-blocage. La section critique de la ressource qui résulte de la fusion correspond à la section critique des deux ressources R_n et R_m. Par ailleurs, le modèle de conception réfutant est constitué de *p-1* ressources et les tâches T_i et T_j partagent seulement R'_n au lieu de R_n et R_m.

L'utilisation de la nouvelle ressource par les deux tâches qui la partagent ($U_{(R'_n,T_i)}$ et $U_{(R'_n,T_j)}$ dépend aussi de la situation au niveau implémentation (Figure 5.13). En effet, pour les deux situations possibles d'inter-blocage la façon de calculer l'utilisation de la ressource résultante n'est pas la même. Pour la première situation (situation 1), l'utilisation de la ressource R'_n par T_i (respectivement par T_j) correspond à la valeur maximale d'utilisation des deux ressources R_n et R_m : $U_{(R'_n,T_i)} = max(U_{(R_n,T_i)}, U_{(R_m,T_i)})$. Par contre pour la deuxième situation (situation 2), l'utilisation de la ressource R'_n par T_i (respectivement par T_j) correspond à la somme d'utilisation des deux ressources $U_{(R'_n,T_i)} = U_{(R_n,T_i)} + U_{(R_m,T_i)}$. Comme nous l'avons déjà mentionné, au niveau conception nous n'avons pas d'idée sur quelle situation est adoptée, nous considérons ainsi le pire cas qui correspond à la deuxième situation. La Figure **??** montre le résultat d'application du patron SRMP dans le but de fusionner les deux ressources R_n et R_m.

Notons que si les tâches partagent plus que deux ressources (p ressources), elles seront

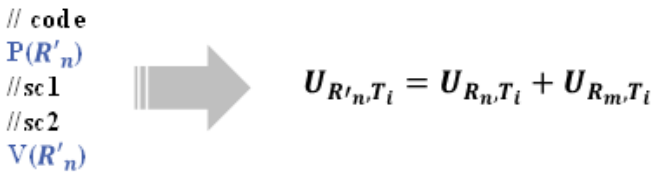


FIGURE 5.14 – *Application du patron SRMP pour la fusion des deux resources R_n et R_m*

fusionnées en une seule ressource. Ainsi, l'utilisation de deux ressources dans la description de la solution de ce patron n'est qu'un exemple qui peut être généralisé pour p ressources. D'autre part, si une autre tâche T_q utilise un des deux ressources R_n et R_m, l'application de ce patron nécessite en plus une mise à jour de l'utilisation de la ressource résultante R'_n par la tâche T_q. Ainsi, la formule générale pour calculer l'utilisation de la nouvelle ressource R'_n par toutes les tâches qui la partagent est donnée comme suit :

$$U_{(R'_n,T_i)} = \sum_{R_j \in shared(T_i) \cap R'_n} U_{R_j,T_i}, \forall T_i \in M_{conception} \tag{5.2}$$

Dans cette expression, le terme $shared(T_i)$ correspond à l'ensemble des ressources utilisées par T_i. Par conséquent, l'utilisation de la ressource résultante de la fusion par une tâche T_i correspond à la somme des utilisations de des ressources qui sont utilisées par T_i d'une part et qui constitue R'_n d'autre part ($shared(T_i) \cap R'_n$). Une description algorithmique de ce patron est donnée par l'algorithme 16.

Cet algorithme définit comme entrée le modèle de conception initial de l'application ($M_{conception}$) et fournit en sortie le modèle de conception modifié ($M_{modifie}$) qui résulte de l'application du patron SRMP. En effet, cet algorithme commence par déterminer s'il existe des tâches qui partagent plus qu'une ressource à la fois. Dans le cas échéant, il fusionne ces ressources en une seule ressource (R_res) puis il met à jour l'utilisation de cette nouvelle ressource par les tâches du modèle résultant ($M_{modifie}$).

Afin de montrer que l'algorithme 16 répond correctement au besoin de l'utilisateur qui est la fusion des ressources partagées par plus qu'une tâche à la fois dans le but d'éliminer une situation potentielle d'interblocage au niveau implémentation, nous considérons l'exemple de modèle de conception initial illustré par la Figure 5.15(a). Nous supposons que ce modèle repose sur une configuration d'une plateforme abstraite d'analyse qui définit un ordre *décroissant* des niveaux de priorité (1 est le niveau le plus prioritaire) et qu'une implémentation basée sur le protocole de synchronisation *PCP* est considérée pour les ressources partagées. D'autre part, nous supposons que le RTOS cible (sur lequel ce modèle sera déployé) ne fournit que *PIP* comme protocole de synchronisation (pas de PCP). Une condition requise pour cet algorithme c'est que toutes les tâches dans le modèle de conception résultant partagent qu'une seule ressource à la fois.

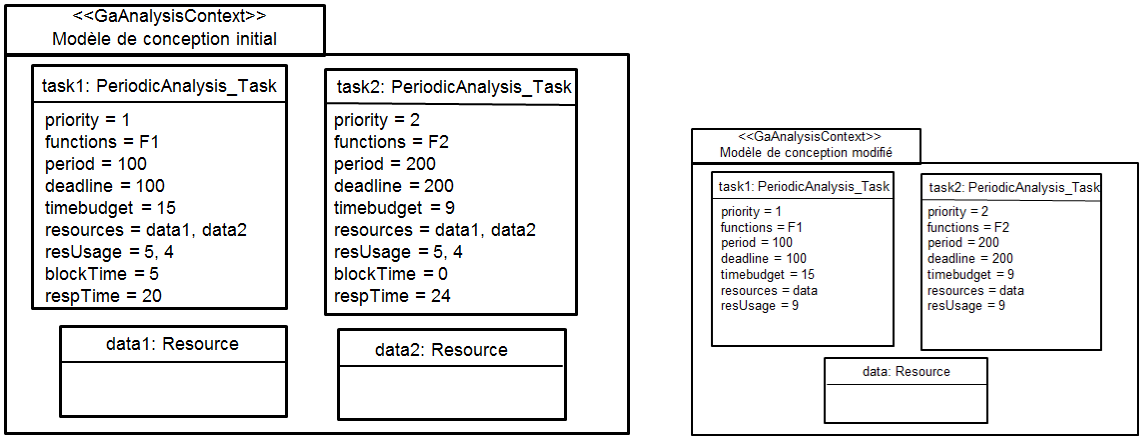
Le modèle de conception initial montre deux tâches *task1* et *task2* qui partagent les deux ressources *data1* et *data2*. Ainsi, le résultat d'application du patron SRMP dans le but d'éli-

Algorithme 16 : Algorithme du patron SRMP

Entrées :
$M_{conception}$: le modèle de conception initial de l'application
Résultat :
$M_{modifie}$: le modèle de conception modifié suite à l'application du patron
Données :
$Tasks$: la liste des tâches dans le modèle de conception initial
$Tasks_{modifie}$: la liste des tâches dans le modèle de conception modifié
$Shared(T_i, T_j)$: la liste des ressources partagées entre les tâches T_i et T_j
$Used - Resource(T_i)$: la liste des ressources utilisées par T_i
R_{res} : la ressource qui résulte de la fusion des autres ressources
$Inter(R_res, Used - Resource(T_i))$: les ressources communes entre une ressource qui résulte de la fusion et les ressources utilisées par une tâche
$Utilization_(R, T_i)$: l'utilisation de la ressource R par la tâche T_i
Initialisations :
$M_{modifie} \leftarrow M_{conception}$;
début
  pour *i* **allant de** *1* **à** *sizeOf(Tasks)-1* **faire**
    pour *j* **allant de** *i* **à** *sizeOf(Tasks)* **faire**
      si $sizeOf(Shared(Tasks.get(i), Tasks.get(j)) > 1)$ **alors**
        $R_{res} = Merge(Shared(Tasks.get(i), Tasks.get(j)))$;
        $M_{modifie} = Update(R_{res})$;
        pour *k* **allant de** *1* **à** $sizeOf(Tasks_{modifie})$ **faire**
          $Utilization_{(R_{res}, Tasks_{modifie}.get(k))} \leftarrow 0$;
          si $Inter(R_{res}, Used - Resource(Tasks_{modifie}.get(k)))$!= **null alors**
            pour *l* **allant de** *1* **à**
            $sizeOf(Inter(R_{res}, Used - Resource(Tasks_{modifie}.get(k)))$ **faire**
              $Utilization_{(R_{res}, Tasks_{modifie}.get(k))} = Utilization_{(R_{res}, Tasks_{modifie}.get(k))} +$
              $Utilization_{Inter(R_{res}, Used - Resource(Tasks_{modifie}.get(k))).get(l), Tasks.get(k)}$;
  Retourner $M_{modifie}$;

TABLE 5.4 – Les caractéristiques de l'algorithme SRMP en termes de complexité et de terminaison

	Calcul de la complexité	**Absence des boucles infinies**
Algorithme du patron SRMP	La complexité de cet algorithme est polynomiale ($o(n^3)$) et dépend du nombre des tâches du modèle de conception initial *(n)*.	Cet algorithme se termine lorsque toutes les tâches du modèle de conception initial ont été traitées (*sizeOf(Tasks)*).



(a) Modèle de conception initial (b) Modèle de conception modifié

FIGURE 5.15 – Exemple d'application du patron SRMP

miner une situation potentielle d'interblocage à l'implémentation est donné par la Figure 5.15(b). Dans ce modèle les deux tâches *task1* et *task2* partagent uniquement la ressource *data* qui résulte de la fusion des deux ressources *data1* et *data2*. Les temps d'utilisation de la nouvelle ressource par les deux tâches sont calculés suivant l'équation 5.2. Suite à l'application de ce patron et la génération du modèle de conception modifié, la phase de refactoring procède à l'analyse de ce modèle (i.e. calcul du temps de blocage et le temps de réponse de chaque tâche) afin de vérifier si les contraintes de temps de l'application sont toujours satisfaites.

Synthèse

Nous avons présenté dans cette section, quatre patrons dans le but de répondre à des problèmes de déploiement différents. Le tableau 5.5 donne pour chaque patron proposé le problème de déploiement qu'il traite, les hypothèses considérées sur le modèle de conception initial et une description du résultat produit par ce patron.

Comme le montre le tableau5.5, nous avons proposé deux patrons (NPAP et DPMP) pour traiter le problème du nombre des niveaux de priorité distincts. Toutefois, si le modèle de conception initial satisfait les hypothèses du patron NPAP, ce patron est par défaut appliqué

pour répondre à ce problème. En effet, contrairement au patron DPMP, l'application du patron NPAP ne fait que modifier le schéma d'affectation des priorités aux différentes tâches en assurant une charge constante du système (l'utilisation du processeur reste constante après l'application de ce patron) [53].

TABLE 5.5 – Description des patrons proposés

Patron	Problème	Hypothèses	Résultat
NPAP	Le nombre des niveaux de priorité distincts utilisé dans le modèle de conception est supérieur à celui autorisé par le RTOS cible pour l'application considérée	Un modèle de conception initial qui consiste de tâches indépendantes ayant des périodes égales à leurs échéances. Ce modèle s'exécute sur une architecture mono-processeur et définit n niveaux de priorités distincts.	Un modèle de conception modifié qui consiste de tâches indépendantes ayant des périodes égales à leurs échéances. Ce modèle définit N niveaux de priorités distincts (N le nombre autorisé par le RTOS pour l'application considérée) .
DPMP	Le nombre des niveaux de priorité distincts utilisé dans le modèle de conception est supérieur à celui autorisé par le RTOS cible pour l'application considérée	Un modèle de conception initial qui consiste de tâches qui peuvent être dépendantes en partageant des ressources (dépendance de données). Ce modèle s'exécute sur une architecture mono-processeur et définit n niveaux de priorités distincts.	Un modèle de conception modifié qui consiste de tâches qui peuvent être dépendantes en partageant des ressources (dépendance de données). Ce modèle définit N niveaux de priorités distincts (N le nombre autorisé par le RTOS pour l'application considérée) .
EPMP	Le modèle de conception initial présente des tâches qui partagent le même niveau de priorité et le RTOS cible, quant à lui, n'autorise pas cette situation	Un modèle de conception initial qui consiste de tâches qui peuvent être dépendantes en partageant des ressources (dépendance de données). Ce modèle s'exécute sur une architecture mono-processeur.	Un modèle de conception modifié qui consiste de tâches qui peuvent être dépendantes en partageant des ressources (dépendance de données) ayant toutes des niveaux de priorité distincts.
SRMP	Le modèle de conception initial est constitué de tâches qui partagent plus qu'une ressource à la fois implémentées en utilisant le protocole de synchronisation *PCP* et le RTOS cible ne fournit quant à lui que *PIP* comme protocole de synchronisation	Un modèle de conception initial qui consiste de tâches qui peuvent être dépendantes en partageant des ressources (dépendance de données). Ce modèle s'exécute sur une architecture mono-processeur.	Un modèle de conception modifié qui consiste de tâches qui peuvent être dépendantes en partageant des ressources. Toutes les tâches de ce modèle partagent au plus une ressource à la fois.

5.5 Conclusion

Une mise en œuvre du processus DRIM a été présentée dans ce chapitre. Cette mise en œuvre explique en plus de détails les différentes étapes du processus DRIM en introduisant à chaque fois les techniques de transformations et les algorithmes utilisés pour la réalisation de chaque étape de ce processus. Cette mise œuvre suppose que le modèle de conception temps réel en entrées décrivant l'application présente uniquement des tâches périodiques s'exécutant sur une architecture mono-processeur et qui ne peuvent être dépendantes qu'en partageant des ressources logicielles.

L'objectif de ce travail est loin de considérer tous les cas possibles lors du déploiement d'une application sur différents RTOS ou de proposer des listes exhaustives des tests de faisabilité du déploiement à réaliser, des patrons, etc. L'intérêt étant d'offrir un cadre méthodologique extensible qui repose sur des standards (UML, MARTE, MDA, etc...) permettant à un utilisateur de l'adapter selon ses besoins (i.e. ajouter des tests, des patrons, es stratégies de portage,etc.).

Afin de montrer l'intérêt de ce processus et son applicabilité sur un cas concret, une automatisation de ce dernier s'avère d'une importance majeure. Cette automatisation se traduit par la proposition d'un outil qui implémente les différentes facettes du processus. Cet outil est présenté dans le chapitre suivant. Nous exposons de même une évaluation de ce dernier et nous discutons les résultats expérimentaux obtenus.

Chapitre 6

Évaluations et expérimentations

Ce chapitre décrit les expérimentations menées dans cette étude pour évaluer le processus DRIM. Ces expérimentations sont basées sur l'automatisation de ce dernier et une présentation d'un cas d'étude permettant ainsi de montrer son applicabilité.

6.1 Introduction

L'objectif de ce chapitre est de présenter une évaluation de la mise en œuvre du processus DRIM proposée dans le chapitre précédent. Cette évaluation repose sur une automatisation de toutes les étapes du processus permettant ainsi de montrer son applicabilité sur un cas concret.

Pour cela, la première partie de ce chapitre introduit le contexte des expérimentations que nous avons menées. La seconde présente un cas d'étude qui servira de support pour montrer l'applicabilité du processus proposé. La troisième partie présente et discute les résultats obtenus.

6.2 Contexte

Initialement, l'objectif de ce travail était de compléter la méthodologie Optimum [62] supportée par l'outil Qompass-Architect (outil CEA), en ajoutant une couche de déploiement du modèle produit par ce dernier sur différents RTOS. Ainsi, nous commençons par introduire rapidement cet outil et plus précisément la méthodologie Optimum associée. Ensuite, nous procédons à la définition de quelques choix afin de permettre l'évaluation de ce processus.

6.2.1 Qompass-Architect

L'objectif de cet outil est de générer automatiquement un modèle de conception (ou de tâches) d'une application temps réel à partir du modèle fonctionnel en suivant la méthodologie Optimum. Comme illustré dans la Figure 6.1, cette méthodologie vise à réduire la distance (*gap*) entre le modèle fonctionnel et le modèle de conception généré. Pour cela, elle introduit la vérification temporelle tôt dans le cycle (à partir du niveau de spécification) afin de guider la génération du modèle de tâches qui satisfait les contraintes temporelles de l'application. Cette méthodologie repose sur deux phases :

- la première phase consiste en une phase d'allocation des budgets de temps aux différentes fonctions du modèle fonctionnel. Cette allocation est guidée par les exigences temporelles de l'application.
- la deuxième phase est une phase d'exploration de l'architecture logicielle. Dans cette phase, Qompass-Architect cherche à produire un modèle d'architecture logicielle qui répond aux exigences temporelles. À la l'issue de cette phase, le modèle de conception obtenu se compose d'un ensemble de tâches, caractérisées par un ensemble de propriétés à savoir la priorité, la période, etc. De plus, chaque tâche du modèle est allouée à un composant matériel spécifique (dans notre cas toutes les tâches sont allouées à un même processeur).

Les expérimentations menées dans cette étude admettent que le modèle de conception initial, point d'entrée de la méthodologie DRIM, est un modèle produit par l'outil Qompass-

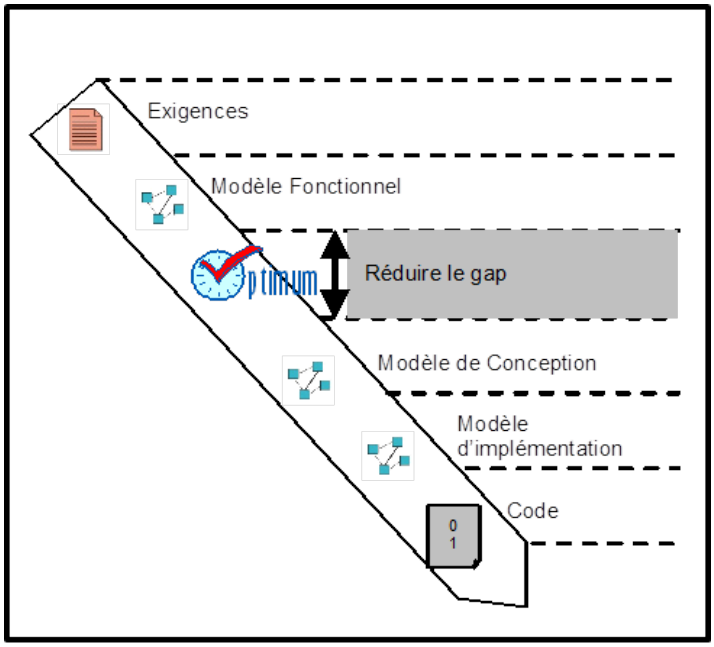


FIGURE 6.1 – *La méthodologie Optimum dans un flot IDM*

Architect. Par conséquent, la plateforme abstraite d'analyse sur laquelle repose ce modèle de conception et dont le modèle constitue aussi une entrée pour le processus DRIM, est une plateforme abstraite associée à cet outil (Qompass-Architect). Ainsi, afin d'évaluer ce processus, une configuration de cette plateforme doit être élaborée en suivant les règles méthodologiques définies dans le chapitre 4 de ce rapport. Un extrait du modèle de cette configuration est donné dans l'annexe B de ce rapport.

6.2.2 Définition des critères d'évaluation

Le but de la méthodologie DRIM est d'assurer le déploiement du modèle de conception initial sur différents RTOS (contexte multiplateforme). Le modèle du RTOS constitue une entrée pour le processus DRIM et doit être élaboré en suivant toujours les règles méthodologiques définies dans le chapitre 4 de ce rapport. Ainsi, afin d'évaluer ce processus, nous avons choisi d'élaborer les modèles de quatre RTOS qui sont RTEMS [8], MicroC-OS/II [10], Nano-Rk [9] (un RTOS pour les réseaux de capteurs sans fils) et AUTOSAR-OS [11]. Des extraits des modèles de ces RTOS sont donnés dans l'annexe B de ce rapport.

Par ailleurs, dans le but de permettre à l'utilisateur d'estimer le coût d'application des différents patrons sur les performances du modèle de conception initial, nous devons fixer les métriques sur lesquelles se base l'étape d'évaluation des performances de la phase du refactoring. Ainsi, les trois métriques considérées pour cette évaluation sont *l'utilisation du processeur*, *le nombre de tâches* et *la robustesse du modèle de conception* que nous expliquons ci-dessous :

– **L'utilisation du processeur**

Pour une architecture monoprocesseur, l'utilisation du processeur est un paramètre important puisqu'il référence la charge du système. En effet, pour un modèle de conception M, l'utilisation du processeur noté U est défini comme suit :

$$\sum_{T_i \in M} \frac{c_i}{P_i} \leq 1 \tag{6.1}$$

Dans cette expression, T_i représente une tâche appartenant au modèle de conception. Par ailleurs, C_i et P_i représentent respectivement le temps d'exécution et la période de la tâche T_i. Une condition de faisabilité nécessaire et suffisante pour un modèle de conception donné (M), est que U doit être inférieure ou égale à 1 ($U \leq 1$)[55].

– **Le nombre de tâches dans le modèle de conception**

Dans certains cas, le nombre de tâches qui réalisent les fonctionnalités d'une application a un impact sur la quantité de la mémoire consommée (en particulier la mémoire RAM) [92]. L'application de certains patrons dans le but de générer le nouveau modèle de conception peut entrainer le changement du nombre de tâches initial. Pour cela, nous avons choisi d'évaluer cette métrique.

– **La robustesse du modèle de conception**

Cette métrique peut être définie comme étant la capacité d'un modèle de conception à respecter toujours les contraintes de temps de l'application, même après avoir changé certains choix de conception (i.e. plus le modèle est robuste, plus la variation des propriétés d'un modèle de conception n'a pas effet sur le respect des propriétés temporelles). Pour un modèle de conception M, nous définissions la robustesse que nous notons ρ_M comme suit :

$$\rho_M = \frac{1}{MinSlack}; MinSlack = min[D_i - Rep_i] \forall T_i \in M \tag{6.2}$$

Dans cette expression, $MinSlack$ correspond à la durée minimale entre l'échéance D_i et le temps de réponse Rep_i de toutes les tâches du modèle [58]. Ainsi, plus la valeur de ρ_M est petite, plus le modèle n'est plus robuste.

6.3 Étude de cas : Application de Contrôle de Robot

Dans cette partie, nous présentons un cas d'étude simple pour illustrer le processus proposé. Cette étude de cas s'appuie sur des briques Lego de type NXT. En effet, le robot est composé d'une brique NXT qui est une brique contrôlée par un ordinateur. Cette brique joue le rôle du cerveau pour le robot. C'est une brique intelligente programmable (Figure 6.2) qui centralise les informations en provenance des capteurs, calcule et fournit des ordres aux moteurs.

Les briques fournissent des éléments matériels intégrés (capteurs, actionneurs, support d'exécution et de communication) et les pilotes de communication avec ces périphériques.

FIGURE 6.2 – *Brique Lego NXT*

Les briques apparaissent ici comme des capteurs intelligents fournissant des services d'acquisition/restitution de données en provenance de capteurs/en direction d'actionneurs.

6.3.1 Aperçu global sur le système étudié

L'architecture matérielle se compose d'une brique NXT que nous appelons Robot et d'un PC de supervision, comme le montre la Figure 6.3. Le PC communique avec le robot via une liaison *Bluetooth*. Une interface homme machine (IHM) est fournie à l'utilisateur pour l'interaction avec le robot.

La brique dispose d'un micro-processeur 32 bits ARM7 d'Atmel de 48 Mhz (256 Ko de mémoire flash et 64 Ko de RAM). Comme tout système embarqué, le robot se compose de capteurs et actionneurs. Le comportement du système est basé sur l'interaction entre ces derniers. Cette brique dispose de deux capteurs : un capteur de luminosité (*Light Sensor*) qui détecte la lumière et mesure son intensité et un capteur d'ultrasons (*UltraSonicSensor*) qui est un capteur de contact. Elle dispose en plus d'un moteur permettant le déplacement du robot et d'une batterie qui alimente ce dernier.

Comme illustré dans la Figure 6.4, le robot est placé sur une feuille blanche dégradée du blanc au noir. L'extrémité blanche de la feuille correspond à la position 0 quant à l'extrémité noire représente la position 100. À la position 100, un obstacle est mis en place indiquant la fin du trajet. Durant son déplacement, le robot calibre son capteur de luminosité pour en faire un capteur de position de 0 (blanc) à 100 (noir). Par conséquent en se basant sur la valeur renvoyée par son capteur de luminosité, l'application de contrôle du PC calcule la position correspondante de 0 à 100. Le capteur d'ultrasons permet de détecter la présence

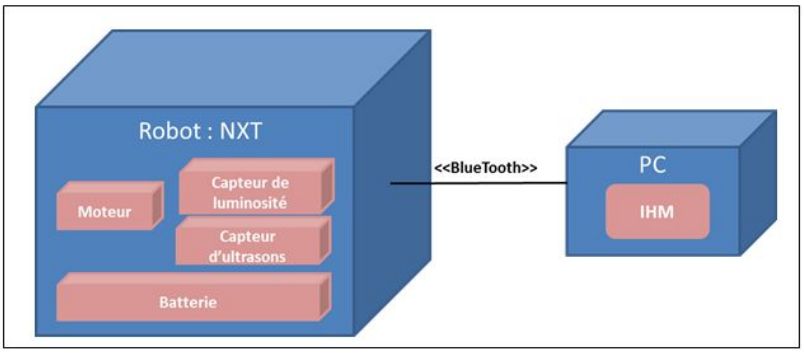


FIGURE 6.3 – *Architecture matérielle cible*

d'un obstacle et mesure la distance le séparant de ce dernier. Ce capteur détecte un objet à moins de 25 centimètres du robot.

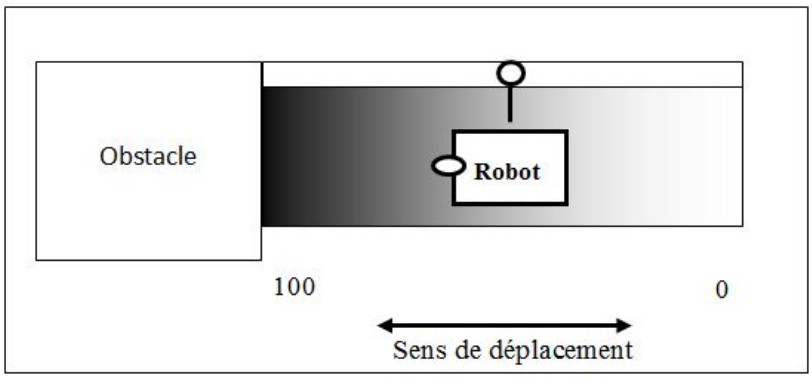


FIGURE 6.4 – *Description du comportement du robot*

Il est à noter que tout le traitement est réalisé au niveau du robot (architecture mono-processeur). Le PC, au travers son interface homme-machine, sert juste pour l'affichage des informations à l'utilisateur.

6.3.2 Spécification de l'application de contrôle de robot

Dans ce paragraphe, nous spécifions les différentes fonctionnalités de l'application de contrôle du robot ainsi que ses contraintes temps réel.

6.3.2.1 Fonctionnement

Cette application définit deux modes : un mode utilisation et un mode interaction. Pour le premier mode, mode utilisation, l'utilisateur a la possibilité de définir une position objectif à travers l'interface IHM. Par conséquent l'application de contrôle du robot doit être en mesure de vérifier cette position objectif par rapport à la position courante et envoyer une commande au moteur si nécessaire pour atteindre la position cible spécifiée par ce dernier.

Dans le mode interaction, le système fournit des informations à l'utilisateur à travers l'interface homme machine IHM. Ces informations concernent : 1) la position courante, 2) la position objectif, 3) le niveau de batterie, et 4) la présence d'un obstacle. Par ailleurs, si un obstacle est détecté, l'application de contrôle envoie une commande au moteur pour l'arrêter.

6.3.2.2 Contraintes temps réel

Cette application est basée sur des événements périodiques. En effet, pour chaque fonctionnalité assurée par cette dernière un événement est associé. Ces événements sont les suivants :

- *Acquisition à partir du capteur de lumière* (*AcquisitionFromLightSensor*) : un événement périodique qui permet de rafraîchir la position courante à partir des données envoyées par le capteur de luminosité.
- *Acquisition à partir du capteur d'ultrasons* (*AcquisitionFromUltraSonicSensor*) : un événement périodique qui permet de vérifier à chaque fois si un obstacle est présent devant le robot.
- *Acquisition à partir de l'utilisateur*(*AcquisitionFromUser*) : un événement périodique qui permet de vérifier si l'utilisateur a demandé la modification de la consigne position qui représente la position cible.
- *Événement de contrôle de la position* (*ControlEvent*) : un événement périodique de contrôle qui permet de vérifier à chaque fois la position courante et la position cible et envoie des commandes au moteur si nécessaire.
- *Événement de contrôle du niveau de batterie* (*PowerControlEvent*) : un événement périodique qui permet d'afficher le niveau de batterie à l'utilisateur.

L'évènement d'acquisition à partir du capteur de lumière pour le rafraîchissement de la position courante est le plus fréquent avec une période de *20 ms*. D'autre part, l'acquisition à partir du capteur ultrasons pour la vérification de la présence d'un obstacle est moins fréquente avec une période de *40 ms*. Les événements d'acquisition à partir de l'utilisateur pour la consigne position et celui de contrôle se font avec la même fréquence avec une période de *100 ms*. Finalement l'événement de contrôle du niveau de batterie est le moins fréquent avec une période de *300 ms*.

De plus, l'échéance de chaque fonctionnalité doit se terminer avant le nouvel événement. Par conséquent, les échéances sont égales aux différentes périodes ($\forall i \in 1..mD_i = P_i$).

6.3.3 Modélisation de l'application de contrôle de robot

Ce paragraphe présente une description fonctionnelle et comportementale de l'application de contrôle de robot en utilisant respectivement le diagramme de structure composite et le diagramme d'activité d'UML.

6.3.3.1 Description fonctionnelle : Diagramme de structure composite

La Figure 6.5 illustre le diagramme de structure composite de l'application de contrôle du robot. Ce diagramme donne une description structurelle des fonctionnalités internes de l'application de contrôle du robot. Cette application est donc représentée par une classe composite (appelée *RobotNXTControl*) encapsulant un ensemble de *parts*. Chaque *part* joue un rôle dans la réalisation des différentes fonctionnalités de l'application et possède un ensemble de *ports* qui matérialisent ses points de connexion. Des connecteurs, dits d'assemblage, permettent de relier les points de connexion entre les différentes *parts*. D'autres connecteurs, dits de délégation, permettent à un port interne de déléguer à un autre port externe (i.e. un port qui permet l'interaction de l'application avec son environnement). Les ports externes peuvent être des ports d'entrées ou de sorties. Les ports d'entrées sont responsables de l'acquisition des données depuis l'environnement externe (acquisition des données depuis les capteurs ou des consignes utilisateur). Tandis que les ports de sorties sont responsables de l'envoie des données à l'environnement externe (des commandes aux actionneurs ou des comptes rendus à l'utilisateur).

Cette application de contrôle de robot reçoit deux types de données : des données de type *SensorData* pour l'acquisition des données depuis les capteurs ou des données de type contrôle *ControlData* . Elle produit deux types de résultat à son environnement : un résultat de type commande pour le contrôle du moteur (*Command* dans la Figure 6.5) ou un résultat de type affichage pour l'utilisateur (*DisplayForUserData* dans la Figure 6.5).

Par exemple, dans la figure 6.5, la fonction *pP* reçoit les données à travers le port *lightData* (responsable de l'acquisition des données à partir du capteur de luminosité) et calcule la position courante en se basant sur l'intensité de la lumière envoyée par ce capteur. Par conséquent, le résultat généré par cette fonction représente la valeur de la position calculée entre 0 et 100 et servira comme entrée pour la fonction *pos* (responsable sur la gestion de la position courante) via le port *inputPosition*. Cette positon est envoyée à l'utilisateur (pour affichage) à travers le port de sortie *outputPositionDisplay*.

6.3.3.2 Description comportementale : Diagramme d'activité

La Figure 6.6 illustre le diagramme d'activité qui donne une description comportementale de haut niveau de l'application de contrôle du robot. Chaque événement, spécifié dans 6.3.2, est représenté par un nœud *Accept Event Action* d'UML (par exemple *acquisitionLightSensor*). Par ailleurs, chaque *part* du diagramme de structure composite est représentée par un nœud *Call Behavior Action* d'UML (par exemple *positionProceBehavior*) permettant de spécifier le comportement interne de chaque fonction (part) par un autre diagramme d'activité ou un *opaque Behavior* (code).

L'outil Qompass-Architect se base essentiellement sur cette description pour la génération d'un modèle de conception qui décrit une réalisation possible de l'application de contrôle de robot tout en assurant le respect des contraintes de temps. Pour cela, ce modèle est stéréotypé «gaWorkloadBehavior» et annoté en plus par les propriétés temporelles

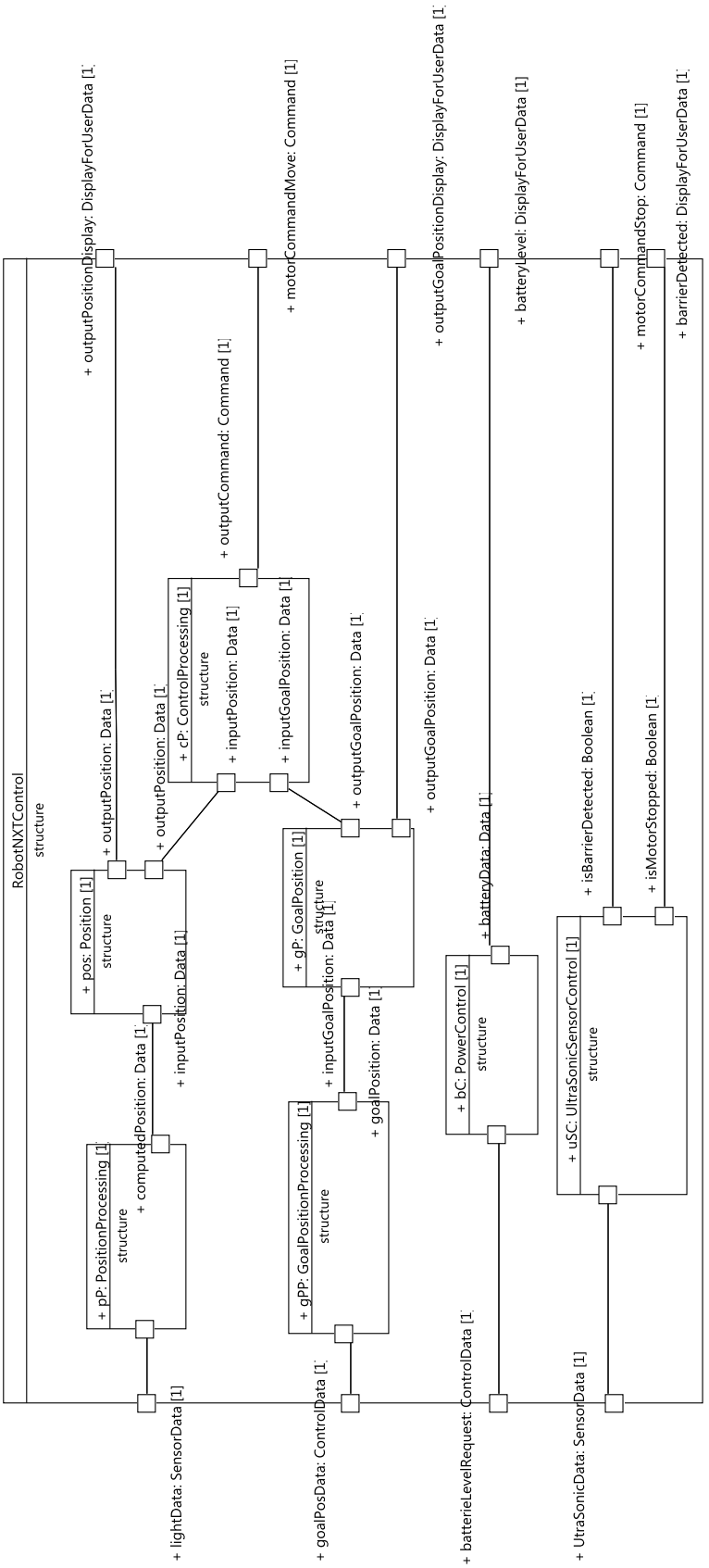


FIGURE 6.5 – *Description fonctionnelle de l'application de contrôle de robot*

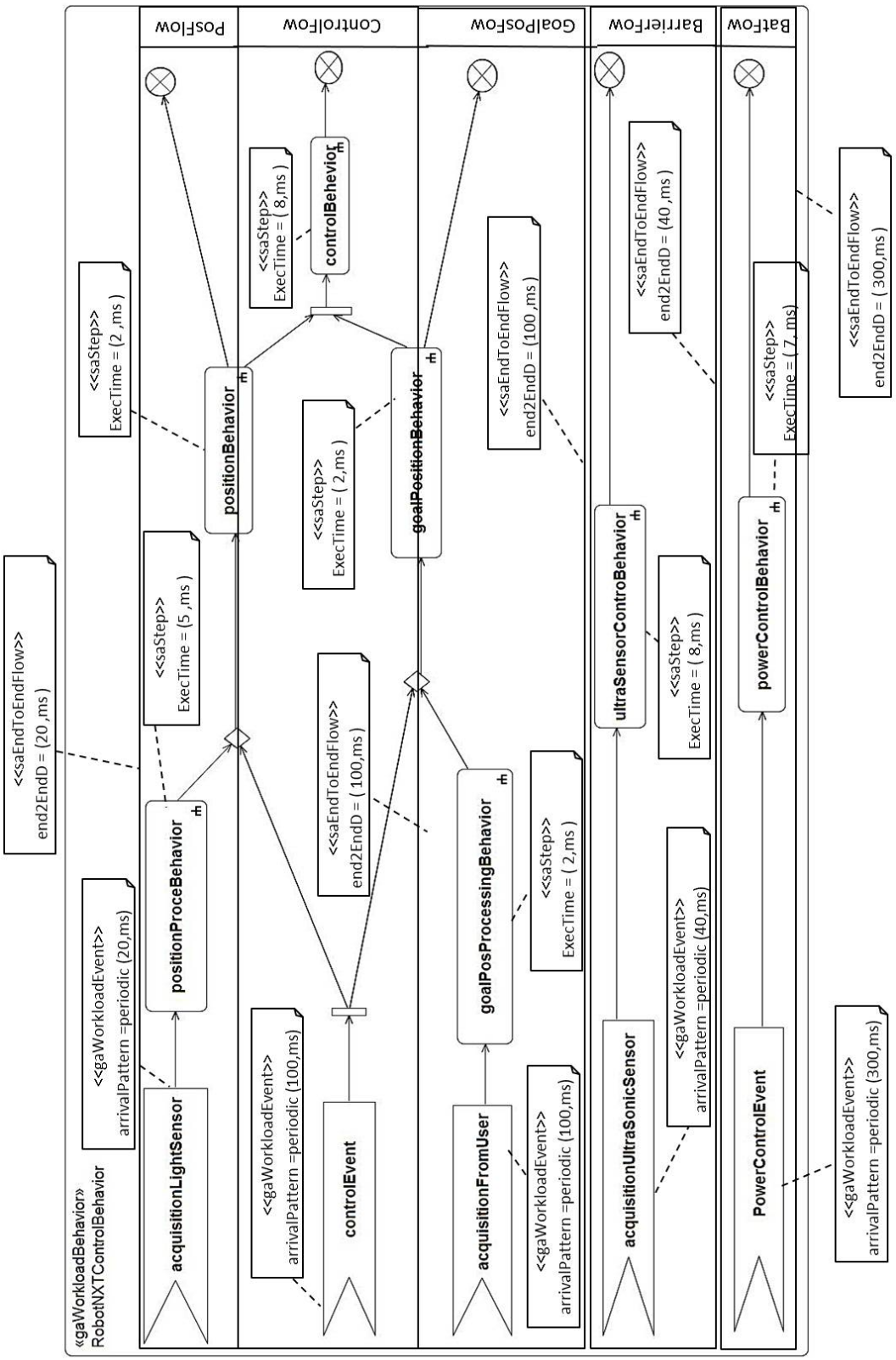


FIGURE 6.6 – *Description comportementale niveau système de l'application de contrôle de robot*

nécessaires pour cette génération. Chaque *Accept Event Action* est annoté par «gaWorkloadEvent» permettant ainsi de spécifier le taux et le type d'activation de l'événement associé. Par ailleurs, chaque *Call Behavior Action* est annoté par «saStep» permettant ainsi de préciser le temps d'exécution du comportement correspondant. La Figure 6.6 montre aussi cinq flux de bout-en-bout (*end-to-end flows*) encapsulants les différents événements et les comportements associés. Chaque flux est annoté par «saEndtoEndFlow» permettant ainsi de spécifier son échéance de bout-en-bout. Partant de ce modèle, nous présentons dans ce qui suit un exemple de modèle de conception initial produit par Qompass-Architect pour l'application de contrôle de robot.

6.3.4 Génération d'un modèle de conception initial pour l'application de contrôle de robot

A partir du modèle de l'application illustré par la Figure 6.6, la génération d'un modèle de conception de l'application en suivant la méthodologie Optimum [62], nécessite en plus la définition de la plateforme sur laquelle se base l'analyse. Cette plateforme décrit, d'une part, l'architecture matérielle de la plateforme cible en termes de nœuds d'exécution et, d'autre part, une configuration de la plateforme logicielle abstraite utilisée pour l'analyse.

La Figure 6.7 montre la plateforme sur laquelle nous nous basons au niveau conception, en particulier pour la génération du modèle de conception initial pour cette évaluation. Cette plateforme appelée *SaPlatform* et annotée par «gaResourcesPlatform», montre le seul nœud d'exécution de l'architecture cible (la brique NXT) *CPU-NXT* ainsi qu'une configuration de la plateforme logicielle abstraite d'analyse (plateforme *Qompass* décrite en annexe B de ce rapport).

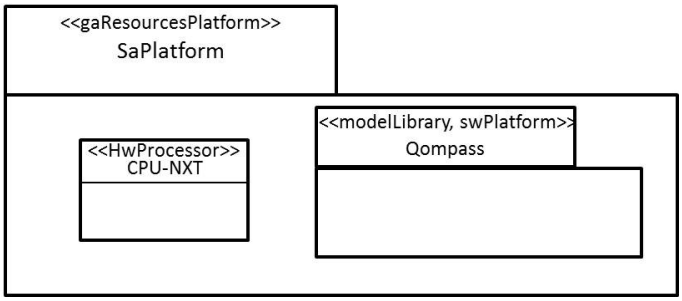


FIGURE 6.7 – *Description de la plateforme utilisée au niveau conception*

Figure 6.8 montre un exemple d'allocation des tâches et des ressources pour l'application de contrôle du robot. Nous avons ainsi identifié cinq tâches : *PositionProcessingTask*, *goalPositionProcessTask*, *controlProcessingTask*, *powerControlTask* et *ultrasonicSensorControlTask* et deux ressources partagées entre ces tâches :*position* et *goalPosition*.

Figure 6.9 donne un exemple de modèle de conception initial pour l'application de contrôle du robot. Comme nous l'avons mentionné dans le chapitre 4 de ce rapport (dans la définition des règles de modélisation des applications *Règle A1*), ce modèle est annoté par

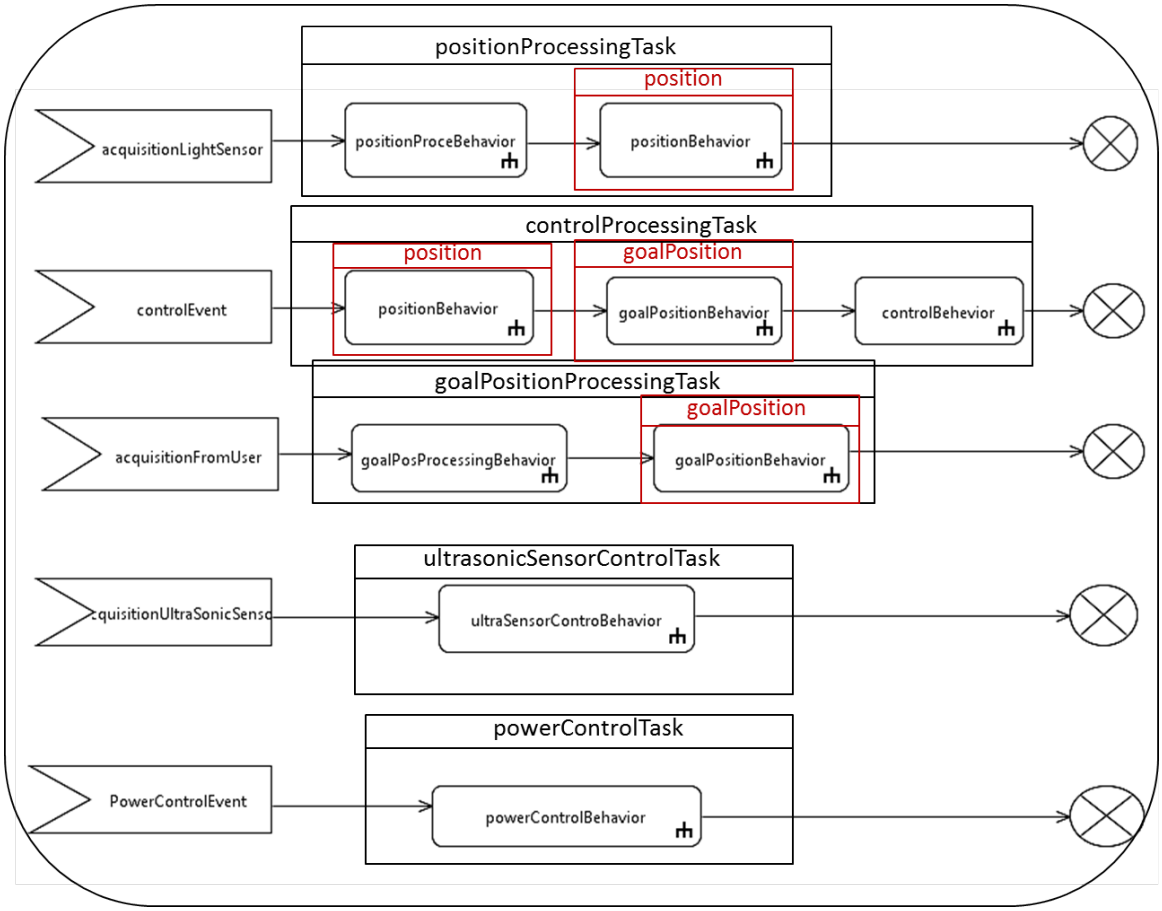


FIGURE 6.8 – *Modèle d'allocation des tâches et des ressources pour l'application de contrôle du robot*

le stéréotype « GaAnalysisContext» de MARTE. En effet, la propriété *wokload* de ce stéréotype référence la description comportementale de l'application donnée par la Figure 6.6. De plus, la propriété *platform* référence la plateforme donnée par la Figure 6.7 utilisée au niveau conception pour la vérification et la génération du modèle de conception. Chaque élément de ce modèle (Figure 6.9) est une instance typée par une ressource définie dans le modèle de la plateforme abstraite d'analyse *Qompass* (présenté en annexe B). Ce modèle se compose ainsi de cinq tâches périodiques : *PositionProcessingTask*, *goalPositionProcessTask*, *controlProcessingTask*, *powerControlTask* et *ultrasonicSensorControlTask* qui sont des instances de la ressource *Qompass_PeriodicTask* définie dans la plateforme abstraite *Qompass*. Ces tâches sont déclenchées respectivement par les événements : *AcquisitionFromLightSensor*, *AcquisitionFromUser*, *ControlEvent*, *PowerControlEvent* et *AcquisitionFromUltraSonicSensor*. Par conséquent, la valeur de la période de chaque tâche correspond à la valeur de la période de l'événement qui l'a déclenché. De plus, le modèle de conception se compose de deux ressources partagées : *position* et *goalPosition*. La ressource *position* qui représente la position courante du robot est partagée entre les deux tâches *PositionProcessingTask* et *controlProcessingTask*. La ressource

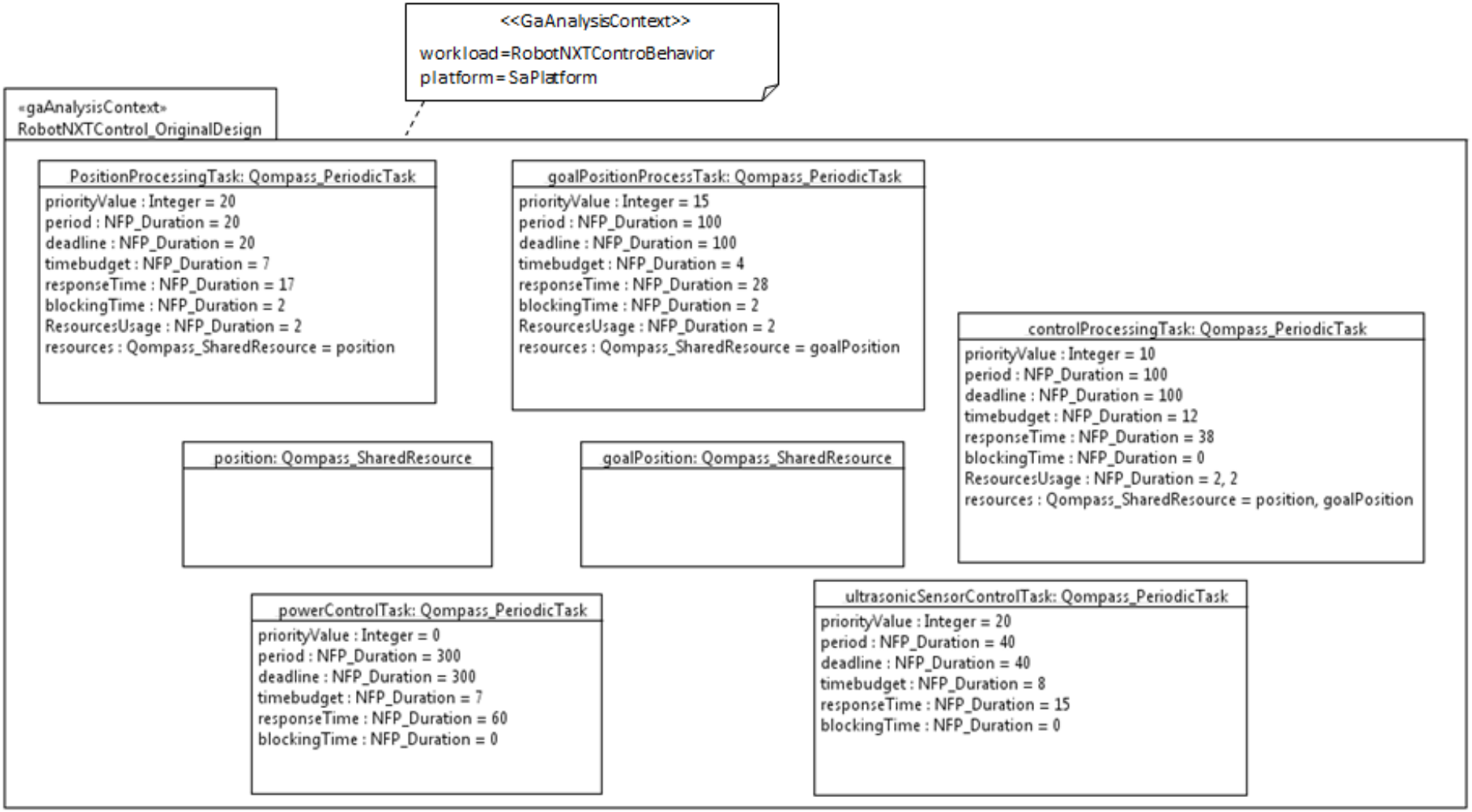


FIGURE 6.9 – *Exemple de modèle de conception initial de l'application de contrôle du robot généré par Qompass-Architect*

goalPosition qui représente la position cible définie par l'utilisateur est partagée par les deux tâches *goalPositionProcessTask* et *controlProcessingTask*. Les choix d'implémentation de ces ressources sont fixés au niveau de la plateforme abstraite d'analyse (*Qompass*) et correspondent à un sémaphore protégé par le protocole de synchronisation PIP pour cette évaluation.

Par ailleurs, pour cette évaluation, la politique d'ordonnancement est fixée à *fixedPriority* au niveau de la plateforme *Qompass* (voir annexe B) (c'est-à-dire une politique à base des priorités fixes), de plus, l'ordre de priorité comme présenté dans la plateforme *Qompass* est croissant. Ainsi, les deux tâches *ultrasonicSensorControlTask* et *PositionProcessingTask* ayant le même niveau de priorité qui est égal à *20* sont les plus prioritaires. Par contre, la tâche *powerControlTask* est la tâche la moins prioritaire avec une valeur de priorité égale à 0.

Notons que ce modèle de conception initial (Figure 6.9) qui sert comme entrée pour le processus DRIM satisfait les hypothèses considérées pour les applications visées par cette étude (toutes les tâches s'exécutent sur une architecture monoprocesseur et ils ne dépendent qu'en partageant des données en exclusion mutuelle).

Le tableau 6.1 donne une représentation, sous forme tabulaire, du modèle de conception initial qui montre les résultats de la vérification temporelle de ce modèle (i.e. les temps de réponse des différentes tâches du modèle). Dans ce tableau :

- P : la période de la tâche
- D : l'échéance de la tâche
- p : la priorité de la tâche
- c : le temps d'exécution de la tâche
- U_R : le temps d'utilisation de la ressource R par la tâche
- B : le temps de blocage de la tâche
- Rep : le temps de réponse de la tâche

TABLE 6.1 – Résultats de vérification temporelle du modèle de conception initial donnés par Qompass

Tâches	P	D	p	c	U_R	B	Rep
PositionProcessingTask	20	**20**	20	7	$U_{position} = 2$	2	**17**
goalPositionProcessTask	100	**100**	15	4	$U_{goalPosition} = 2$	2	**28**
controlProcessingTask	100	**100**	10	12	$U_{position} = 2$ $R_{goalPosition} = 2$	0	**38**
ultrasonicSensorControl Task	40	**40**	20	8	-	0	**15**
powerControlTask	300	**300**	0	7	-	0	**60**

A partir de ce tableau, nous pouvons remarquer que le temps de réponse de chaque tâche dans le modèle de conception initial est inférieur à son échéance. Par la suite, ce modèle respecte les contraintes de temps de l'application de contrôle du robot.

6.4 Résultats et discussion

L'objectif de cette partie est de discuter les résultats de déploiement du modèle de conception initial de l'application de contrôle de robot, présenté précédemment, en suivant le processus DRIM. En d'autres termes, nous nous intéressons à évaluer la capacité de l'outil Qompass-Architect à guider l'utilisateur lors de ce déploiement. Pour cela, nous avons identifié deux cas : Le premier cas correspond à un déploiement sur un RTOS qui peut varier (l'utilisateur n'a pas de contraintes par rapport au RTOS cible). Par contre, dans le second cas, nous supposons que l'utilisateur ne peut pas changer de RTOS (dans ce cas le déploiement est réalisé sur un RTOS qui est imposé). Ces deux cas seront détaillés dans les parties qui suivent.

6.4.1 Déploiement de l'application sur un RTOS qui peut varier

L'objectif de l'utilisateur est de générer, à partir du modèle de conception initial précédemment présenté, un modèle de l'application de contrôle de robot spécifique à un RTOS. Dans ce cas, si un problème de déploiement apparait pour un RTOS donné, la tâche de l'outil est de guider l'utilisateur à choisir un autre RTOS afin d'éviter ce problème. Pour cela, l'utilisateur commence par choisir un RTOS à partir de la librairie des modèles constituée des quatre RTOS dont les modèles présentés en annexe B (voir Figure 6.10).

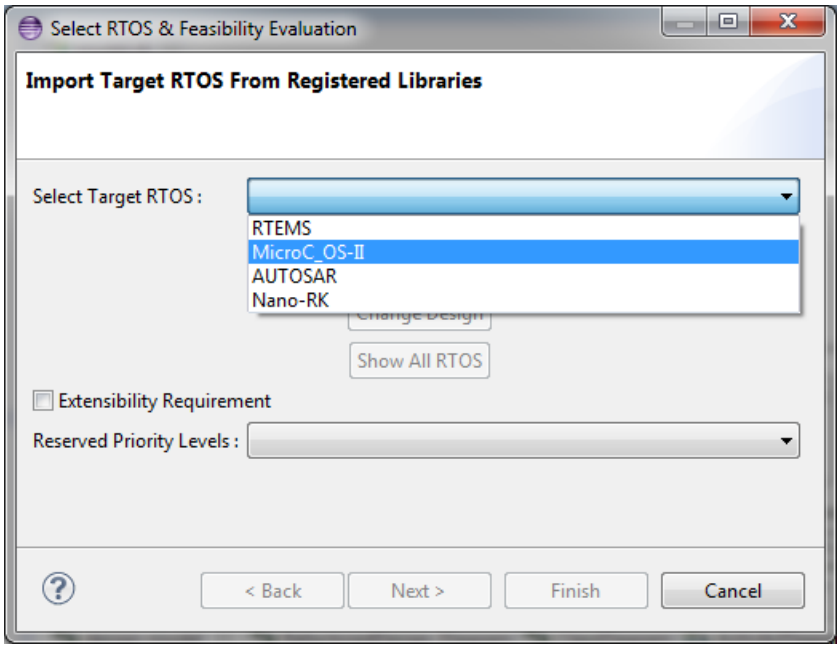


FIGURE 6.10 – *Sélection d'un RTOS cible à partir d'une librairie de modèle*

Supposons que l'utilisateur choisit MicroC-OS/II comme RTOS cible, l'outil procède à une évaluation de la faisabilité du déploiement du modèle de conception initial (de l'ap-

plication du robot) sur ce RTOS. Comme illustré dans la Figure 6.11, la phase d'évaluation de faisabilité détecte deux problèmes de déploiement dans ce cas. Le premier problème est produit par le test des niveaux de priorités égales (algorithme 4 du chapitre 4), le second est produit par le test des ressources partagées (algorithme 2 du chapitre 4) .En effet, comme décrit dans son modèle donné en annexe B, MicroC-OS/II n'autorise pas le partage d'un même niveau de priorité entre les tâches (*isPriorityShared* est égal à *false*) et ne supporte aucun protocole de synchronisation pour l'implémentation des ressources partagées. Tandis que dans le modèle de conception initial (Figure 6.9)les deux tâches *positionProcessingTask* et *ultrasonicSensorControlTask* ont le même niveau de priorité qui est égal à *20* d'une part et d'autre part l'implémentation des deux ressources partagées *position* et *goalPosition* se base sur l'utilisation du protocole de synchronisation *PIP*.

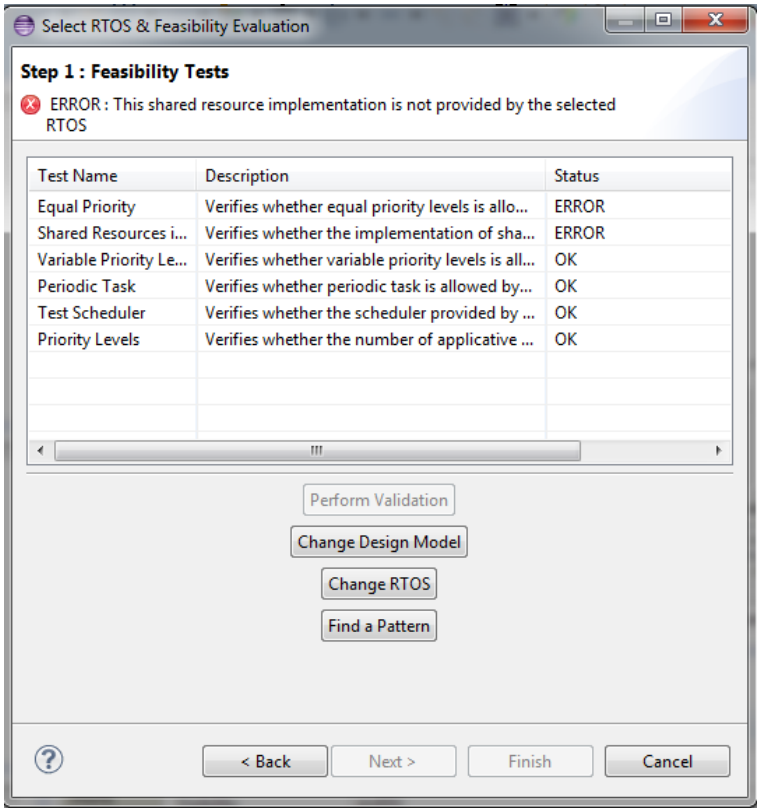


FIGURE 6.11 – *Évaluation de faisabilité du modèle de conception initial pour MicroC-OS/II*

Afin d'éviter ces deux problèmes, l'utilisateur choisit de changer ce RTOS. Dans ce cas, *Qompass-Architect* implémente un filtre qui permet de guider l'utilisateur à choisir un RTOS (à partir de la librairie de RTOS) qui ne pose pas ces deux problèmes. Ceci nous amène à identifier des familles de RTOS suivant un problème de déploiement donné. Par exemple, pour le problème de déploiement lié aux niveaux de priorité égaux, nous identifions une famille de RTOS qui supporte le partage d'un même niveau de priorité entre les tâches (*support_Equal*)

et une famille qui ne supporte pas une telle situation (*notSupport_Equal*). De même pour l'implémentation des ressources partagées, nous identifions une famille de RTOS qui supporte le protocole de synchronisation PIP (*support_PIP*) et une famille qui ne le supporte pas (*notSupport_PIP*). En se basant sur cette classification, ce filtre permet d'éviter pour cet exemple tous les RTOS (à partir de la librairie de modèles) qui n'autorisent pas le partage d'un même niveau de priorité (c'est-à-dire qui appartiennent à la famille *notSupport_Equal*) et qui ne fournissent pas le protocole de synchronisation PIP (c'est-à-dire appartiennent à la famille *notSupport_PIP*). Le tableau 6.2 ci-dessous donne une classification des différents RTOS considérés pour cette évaluation selon les familles identifiées par rapport au problème de déploiement lié aux niveaux des priorités égales et le choix du protocole de synchronisation pour l'implémentation des ressources partagées :

TABLE 6.2 – Classification des RTOS selon les familles identifiées

RTOS	Support_Equal	notSupport_Equal	Support_PIP	notSupport_PIP
MicroC-OS/II		✓		✓
RTEMS	✓		✓	
Nano-RK	✓			✓
AUTOSAR-OS	✓			✓

Ce tableau justifie le résultat d'application du filtre pour l'exemple considéré et qui est illustré par la Figure 6.12 ci-dessous. En effet, tous les RTOS de la librairie de modèles sauf RTEMS mènent potentiellement au moins à un des deux problèmes de déploiement mentionnés (c'est-à-dire appartiennent à *notSupport_Equal* et/ou *notSupport_PIP*). Ainsi, l'application de ce filtre permet de garder juste RTEMS et d'éliminer tous les autres RTOS (c'est-à-dire Nano-RK, AUTOSAR-OS et MicroC-OS/II) afin d'éviter l'apparition des mêmes problèmes de déploiement.

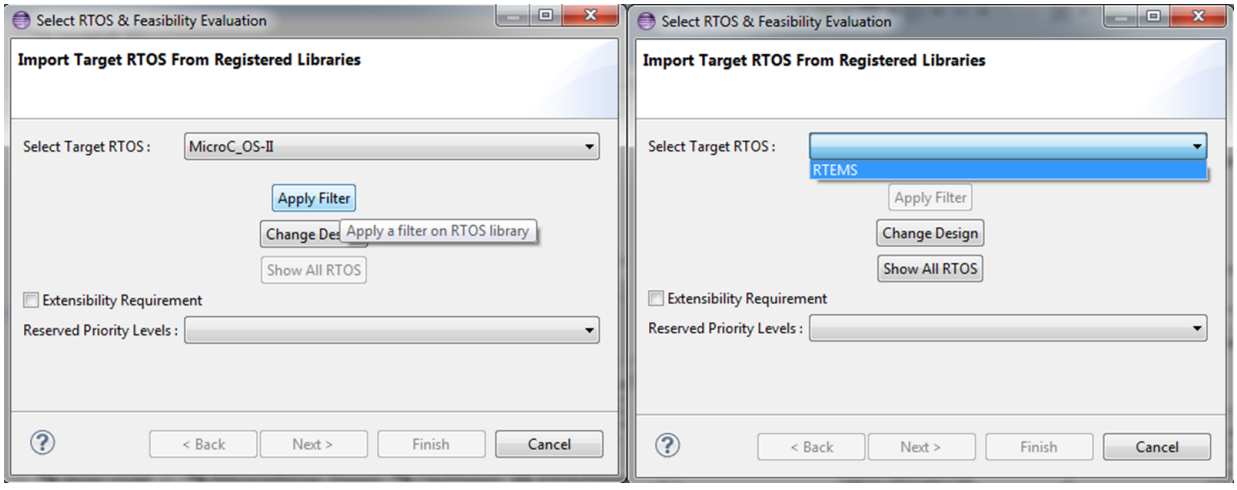


FIGURE 6.12 – *Application du filtre pour guider l'utilisateur à choisir le RTOS approprié*

6.4.2 Déploiement de l'application sur un RTOS fixe

Dans cette partie, nous nous intéressons au cas où le RTOS cible sur lequel l'application sera déployée, est fixe. En d'autres termes, si un problème de déploiement apparait pour le RTOS considéré, l'utilisateur ne peut pas changer de RTOS. Dans ce cas, *Qompass-Architect* doit être capable de guider l'utilisateur pour changer le modèle de conception initial de façon à résoudre le problème de déploiement d'une part et en assurant toujours le respect des contraintes de temps de l'application d'autre part.

Pour cela, supposons dans ce cas que l'utilisateur choisit RTEMS comme RTOS cible sur lequel il cherche à déployer l'application de contrôle du robot. De plus, pour cet exemple, supposons qu'il choisit de limiter le nombre de niveaux des priorités distincts pour des besoins d'extensibilité. Ainsi, il réserve juste *3 niveaux* de priorité distincts pour l'application de contrôle de robot à l'implémentation (voir Figure 6.13). Pour un tel scénario, comme illus-

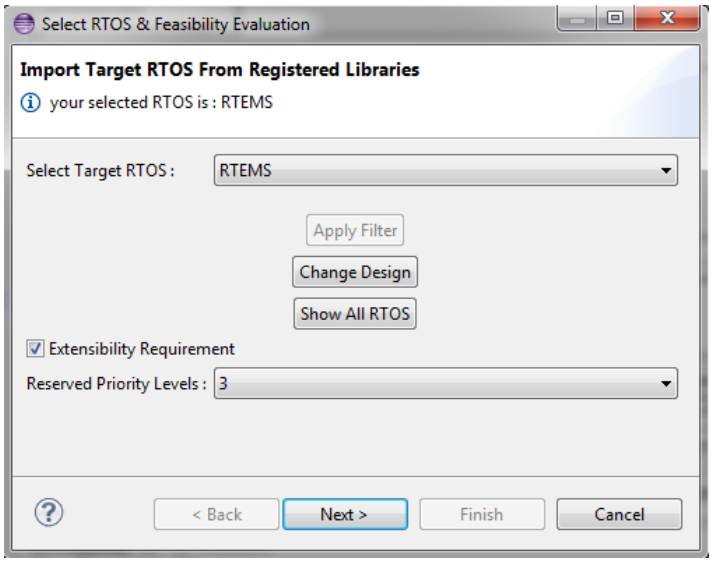


FIGURE 6.13 – *Sélection du RTOS cible et limitation du nombre des niveaux de priorité distincts pour des besoins d'extensibilité*

tré dans la Figure 6.14, l'évaluation de faisabilité du déploiement détecte un problème pour le nombre des niveaux de priorité distincts (produit par le test décrit par l'algorithme 3 du chapitre 4). En effet, le modèle de conception initial définit quatre niveaux de priorité distincts pour décrire l'application de contrôle de robot, tandis que l'utilisateur veut réserver juste trois niveaux pour cette application à l'implémentation.

Dans ce cas, *Qompass-Architect* permet à l'utilisateur de changer le modèle de conception initial manuellement (bouton *Change Design Model* de la Figure 6.14) ou automatiquement en appliquant des patrons (bouton *Find Pattern* de la Figure 6.14) dans le but d'éliminer le problème de déploiement signalé.

Si l'utilisateur choisit de chercher un patron, une liste des patrons qui correspond au pro-

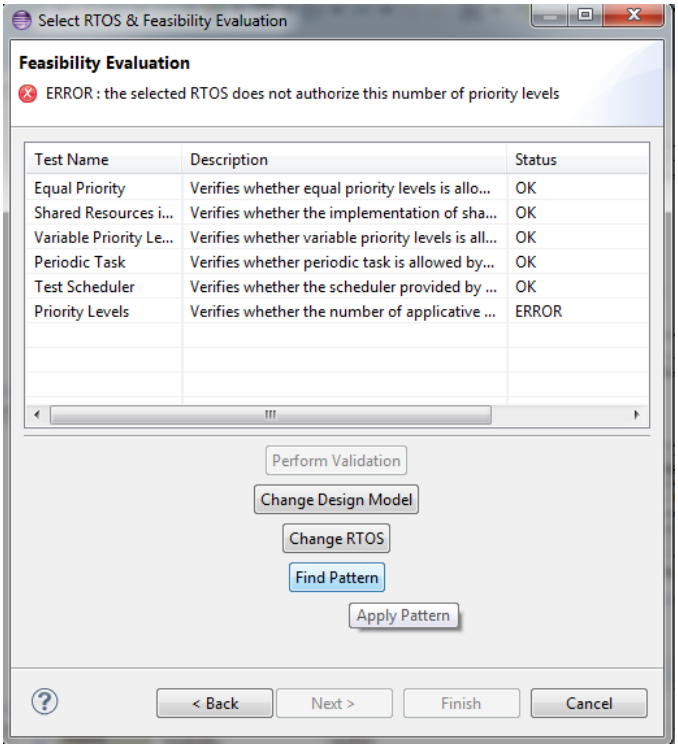


FIGURE 6.14 – *Évaluation de faisabilité du déploiement du modèle de conception initial pour RTEMS en limitant le nombre des niveaux de priorité distinct à 3*

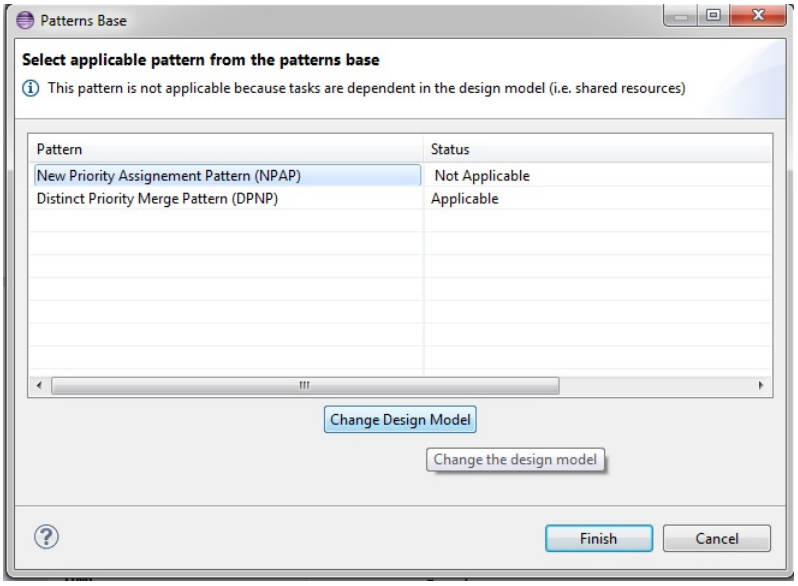


FIGURE 6.15 – *Liste des patrons proposée par l'outil* Qompass *lié au problème de déploiement du nombre des niveaux de priorité distincts*

blème de déploiement lié au nombre des niveaux de priorité distincts apparait comme illustré dans la Figure 6.15. Cette liste contient les deux patrons décrits dans le chapitre précédent et qui sont *New Priority Assignement Pattern (NPAP)* et *Distinct Priority Merge Pattern (DPMP)*. Notons que le patron NPAP n'est pas applicable dans ce cas car le modèle de conception initial ne satisfait pas les hypothèses définies par ce patron (les tâches dans le modèle de conception initial sont dépendantes). Ainsi, seul le patron DPMP peut être appliqué.

L'application du patron DPMP se traduit par la génération d'un nouveau modèle de conception comme le montre la Figure 6.16.

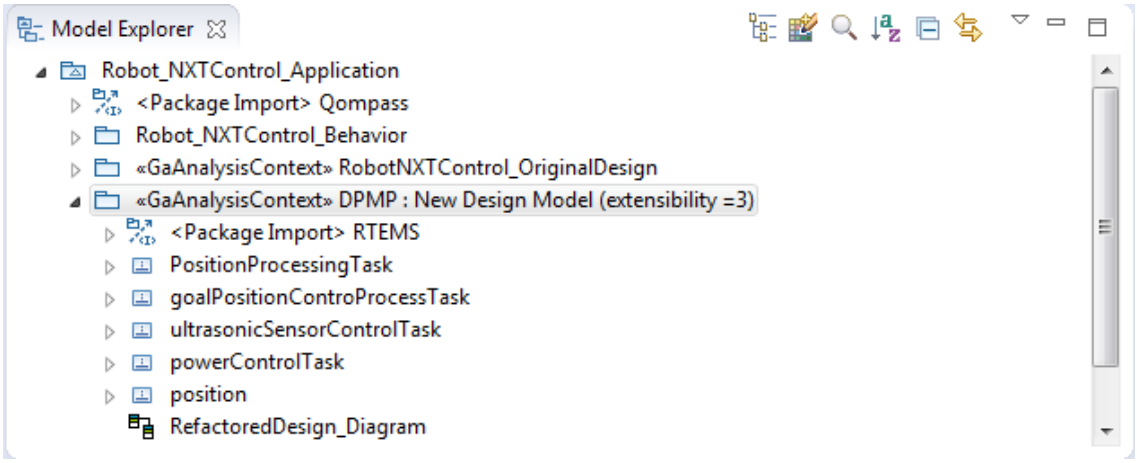


FIGURE 6.16 – *Génération d'un nouveau modèle de conception en appliquant le patron DPMP*

Comme nous l'avons expliqué dans le chapitre précédent, l'application du patron DPMP fait appel à un solveur et se traduit par l'exécution d'un programme linéaire qui décrit ce patron. Ce programme linéaire permet pour ce scénario (l'application de contrôle de robot, RTEMS et degré d'extensibilité 3) de chercher la meilleure façon de fusionner les tâches (en termes d'utilisation processeur) afin de réduire le nombre de niveaux de priorité distincts utilisé dans le modèle de conception initial de 4 à 3. Pour cela, le solveur décide de fusionner les deux tâches *goalPositionProcessTask* et *controlProcessingTask* en une seule tâche *goalPositionControProcessTask* (voir Figure 6.16). Ainsi, le modèle résultant (appelé *DPMP : New Design Model (extensibility=3)*) est constitué de quatre tâches définies avec trois niveaux de priorité distincts et d'une seule ressource partagée qui est *position* (la ressource *goalPosition* est supprimée car elle n'est partagée que par les deux tâches *goalPositionProcessTask* et *controlProcessingTask* qui ont été fusionnées). Une description schématique du modèle résultant est donnée par la Figure 6.17.

Comme nous l'avons précédemment signalé, suite à l'application d'un patron, une vérification temporelle du modèle produit doit être effectuée. Le tableau 6.3 donne une représentation, sous forme tabulaire, du modèle de la Figure 6.17 qui montre les résultats de la vérification temporelle de ce modèle. Ce tableau montre bien que le modèle de conception généré suite à l'application du patron DPMP satisfait les contraintes de temps de l'application de contrôle de robot (i.e. les temps de réponse de toutes les tâches du modèles sont

TABLE 6.3 – Résultats de vérification temporelle du nouveau modèle de conception produit suite à l'application du patron DPMP

Tâches	P	D	p	c	U_R	B	Rep
PositionProcessingTask	20	**20**	20	7	$U_{position} = 2$	2	**17**
goalPositionControProcess Task	100	**100**	15	16	$U_{position} = 2$	2	**40**
ultrasonicSensorControl Task	40	**40**	20	8	-	0	**15**
powerControlTask	300	**300**	0	7	-	0	**60**

inférieures à leurs échéances).

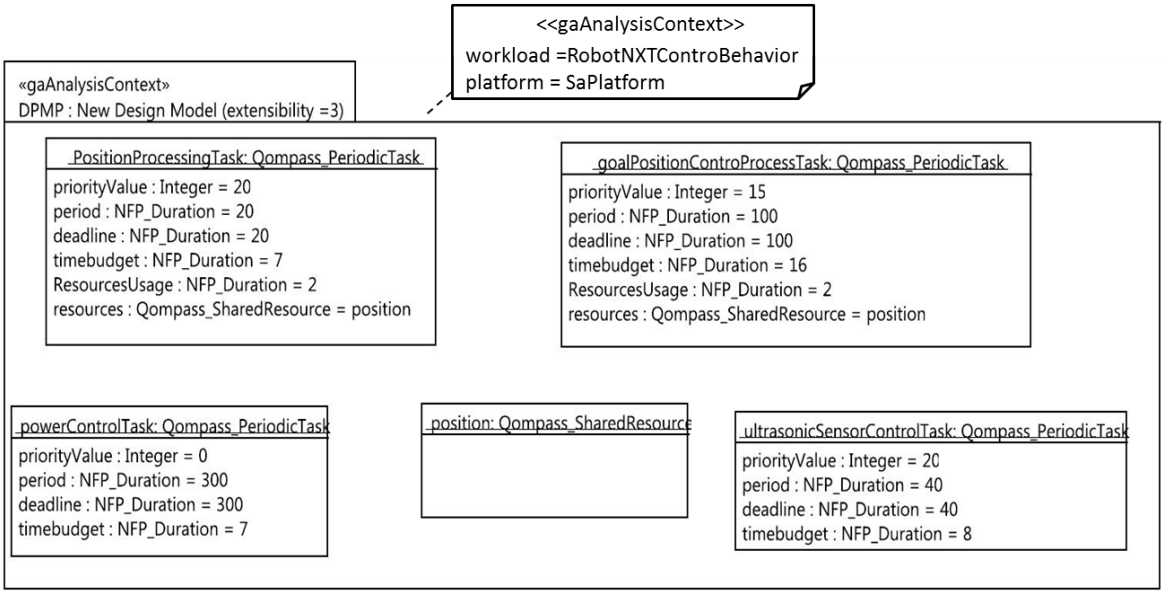


FIGURE 6.17 – *Modèle de conception de l'application de contrôle de robot généré suite à l'application du patron DPMP*

La Figure 6.18 montre une comparaison en termes des métriques de performance (fixées dans le paragraphe 6.2.2 de ce chapitre) entre le modèle de conception initial de l'application de contrôle de robot (Figure 6.9) et le nouveau modèle de conception (Figure 6.17). Nous pouvons remarquer que la charge du système (l'utilisation du processeur) augmente après l'application du patron DPMP (pour le nouveau modèle de conception). En effet, l'objectif d'appliquer les différents patrons dans notre approche est résoudre des problèmes de déploiement pour un RTOS donnée. Par conséquent, l'application d'un patron peut être à l'origine d'une perte de performances qui se traduit par un coût de déploiement. Ceci nous amène à parler du coût de déploiement d'une application (décrite par un modèle de conception initial) sur une famille de RTOS donnée (un exemple d'évaluation de ce coût pour différents RTOS est donné en annexe A) ou lorsque des contraintes d'implémentation

supplémentaires se rajoutent (par exemple ici l'extensibilité).

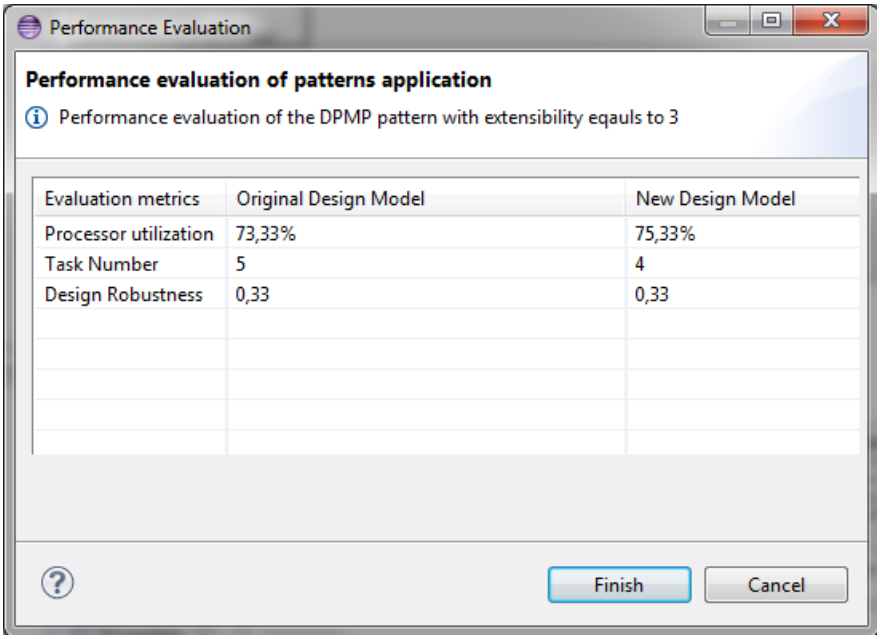


FIGURE 6.18 – *Évaluation des performances du modèle de conception généré suite à l'application du DPMP*

L'évaluation de faisabilité de déploiement de ce nouveau modèle pour RTEMS ne produit aucune erreur. Ainsi, à partir de ce modèle de conception, l'utilisateur a la possibilité de passer à la phase de portage qui permet de générer le modèle de l'application de contrôle de robot spécifique à RTEMS. Comme le montre la Figure 6.19, plusieurs modèles de l'appli-

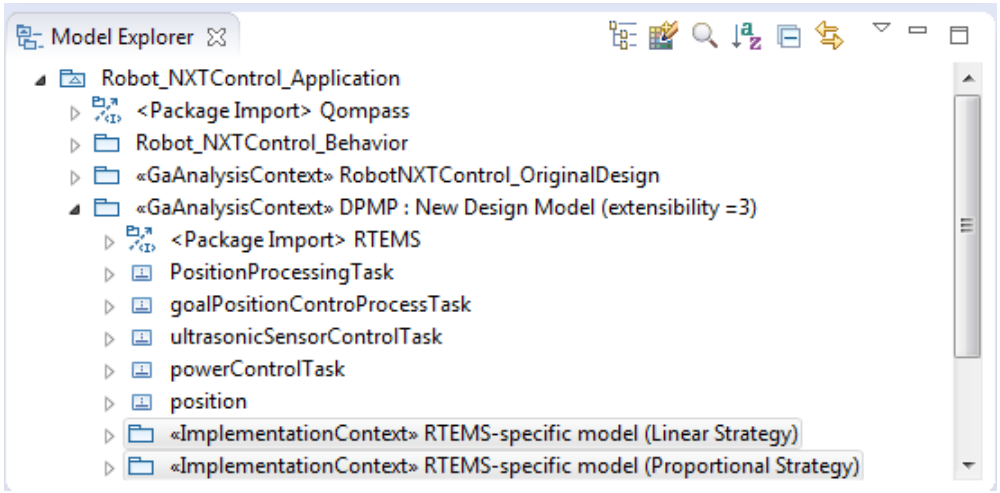


FIGURE 6.19 – *Génération des modèles de l'application de contrôle de robot spécifiques à RTEMS*

cation de contrôle de robot spécifiques à RTEMS peuvent être générés selon la stratégie de

portage des valeurs de priorités choisies.

Figure 6.20 donne une description schématique du modèle de l'application de contrôle de robot spécifique à RTEMS généré en utilisant une stratégie linéaire pour le portage des valeurs des priorités. Ce modèle consiste en quatre tâches *PositionProcessingTask*, *goalPositionControProcessTask*, *powerControlTask* et *ultrasonicSensorControlTask* qui sont des instances du type *rtems_periodicTask* décrit dans le modèle de la plateforme RTEMS (donné en annexe B). En effet, *rtems_periodicTask* est le type approprié qui correspond à *Qompass_PeriodicTask* puisqu'il satisfait les conditions de portage expliquées dans 5.3.1.1 du chapitre précédent. Par ailleurs ce modèle définit une ressource partagée *position* instance du type *PIP_Semaphoe_Resource* décrit aussi dans le modèle de RTEMS et qui définit la même implémentation de la section critique choisie dans le modèle de conception source (spécifié dans le modèle de la plateforme abstraite *Qompass* donné en annexe B).

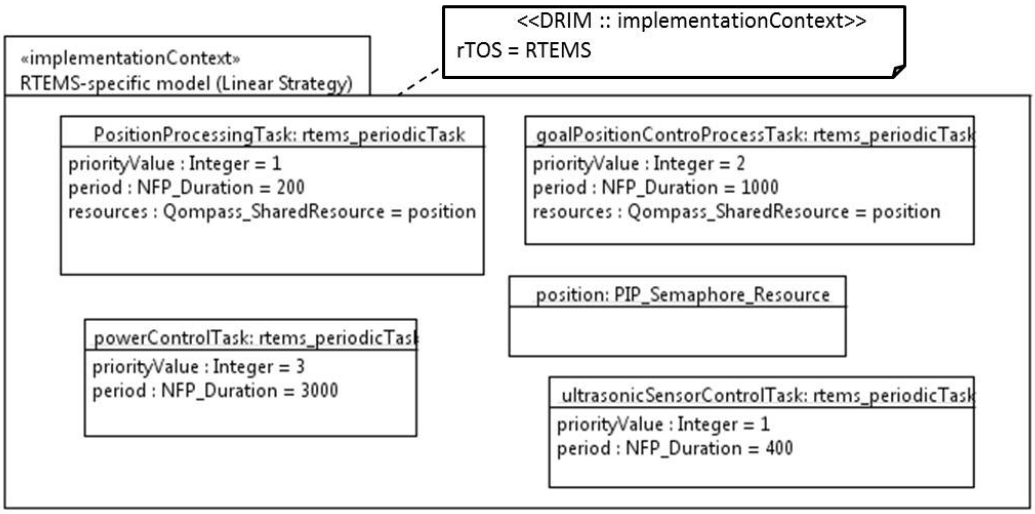


FIGURE 6.20 – *Un modèle de l'application de contrôle de robot spécifique à RTEMS*

De plus, l'algorithme de portage (algorithme 6 du chapitre précédent) permet d'identifier les propriétés qui nécessitent un portage de leurs valeurs (en se basant sur les règles de correspondances des propriétés) et qui sont *priorityValue*, *period* et *ressources* dans ce cas. La Figure 6.20 montre bien que les valeurs des priorités des tâches à l'implémentation suivent une stratégie linéaire puisque elles sont consécutives. De plus, les valeurs des périodes sont exprimées en *tick* dans le modèle résultant par contre elles sont en *ms* dans le modèle de conception source. En effet, la valeur du *clock tick* est fixée à *100 µs* dans le modèle de RTEMS, ce qui explique cette différence entre les valeurs des périodes du modèle source (modèle de conception) et du modèle destination (modèle spécifique à RTEMS).

La dernière phase de processus DRIM est la phase de validation de portage qui permet de confirmer si le modèle généré correspond bien au modèle de conception source en vérifiant un ensemble des propriétés. La vérification des propriétés expliquées dans 5.3.2 du chapitre précédent pour le modèle de l'application de contrôle de robot spécifique à RTEMS

(Figure 6.20) est donnée par la Figure 6.21. Cette Figure montre que toutes les propriétés sont vérifiées et par la suite la phase de portage est réalisée avec succès. Ceci nous permet de confirmer que les propriétés temporelles vérifiées par Qompass au niveau conception restent respectées à l'implémentation du fait que le modèle de conception (Figure 6.17 et le modèle spécifique à RTEMS (Figure 6.20) qui décrivent l'application de contrôle du robot sont sémantiquement équivalent.

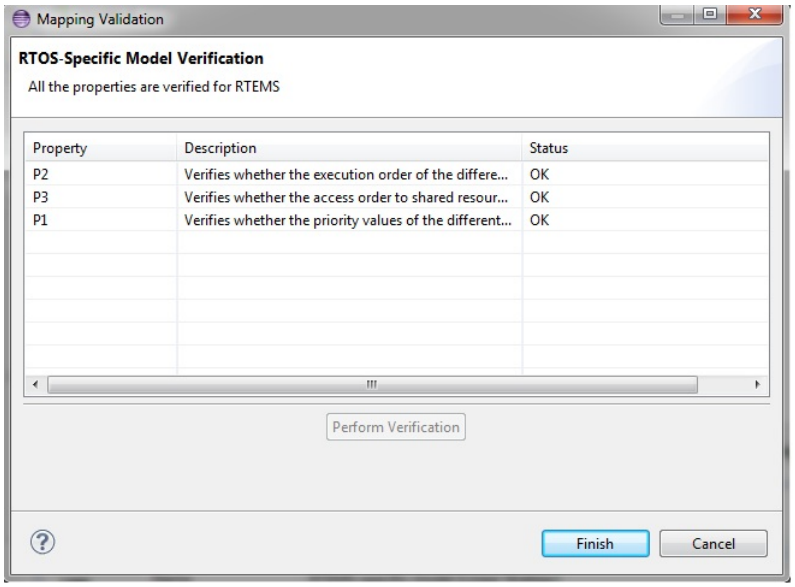


FIGURE 6.21 – *Vérification du modèle spécifique à RTEMS (Validation du portage)*

6.5 Conclusion

Dans ce chapitre, une évaluation de la mise en œuvre du processus DRIM définie dans le chapitre précédant a été proposée. Pour cela, un outil (*Qompass-Architect*) qui implémente les différentes étapes du processus a été réalisé. Un cas d'étude qui consiste en une application simple de contrôle de robot a été également exposé. L'intérêt de cette évaluation est d'examiner l'applicabilité du processus sur un cas concret d'une part et de montrer sa capacité à guider un utilisateur lors du déploiement de son application temps réel sur un RTOS (en visant différents RTOS) d'autre part.

L'outil développé permet la détection des modèles de conception non-implémentables pour un RTOS donné. Ceci permet d'éviter les déploiements non-faisables menant ainsi à des cycles de développement allongés (dans le but de réduire le temps de mise sur le marché). Cet outil permet aussi, dans ce dernier cas, de guider l'utilisateur pour changer de RTOS ou pour modifier le modèle de conception initial afin d'éliminer les problèmes potentiels après le déploiement. Par ailleurs, nous avons constaté suite aux expérimentations menées que la portabilité de l'application (décrite par un modèle de conception qui satisfait les

contraintes de temps de celle-ci) est toujours garanti. En effet, même si le modèle de conception est parfois modifié dans le but de résoudre un problème de déploiement pour un RTOS donné, les changements effectués assurent toujours que ce modèle reste indépendant de ce dernier (c'est-à-dire de le RTOS considéré). Enfin, l'outil réalisé montre une grande flexibilité en permettant la génération de plusieurs modèles spécifiques à différents RTOS (même aussi à un même RTOS selon différentes stratégies). En dépit de cette flexibilité, *Qompass-Architect* garantit des déploiements corrects en vérifiant la conformité des modèles générés (modèle spécifiques à des RTOS) par rapport au modèle de conception initial. Ceci comble l'écart entre la phase de conception et celle d'implémentation et assure que les propriétés temporelles déjà vérifiées au premier niveau (conception) restent préservées à l'implémentation.

Chapitre 7

Conclusion et perspectives

7.1 Bilan

Les travaux présentés dans ce document s'intéressent au développement des systèmes temps réel embarqués dans le cadre de l'ingénierie dirigée par les modèles. Plus précisément, l'objectif de cette étude était d'assurer le déploiement d'une application temps réel sur différents systèmes d'exploitation temps réel (RTOS). A cette fin, le MDA préconise une description architecturale de l'application d'une manière indépendante de toutes technologies logicielles durant la phase conception. Par la suite, une activité d'analyse permet de vérifier si cette description respecte les contraintes de temps imposées par l'application temps réel. De ce fait, le déploiement d'une application temps réel correspond à une transformation de portage ayant comme entrées la description de celle-ci réalisée au niveau conception et le modèle de l'RTOS cible permettant ainsi de générer plusieurs modèles de l'application spécifiques à différents RTOS.

Une synthèse de l'état de l'art a montré que tous les travaux qui se sont intéressés au développement des applications temps réel en suivant une approche MDA considèrent, lors de la phase conception, des hypothèses sur la plateforme logicielle cible (RTOS) afin de permettre la vérification des proporiétés temporelles. En effet, ces hypothèses sont en général intégrées d'une façon implicite dans le modèle de l'application. Cette pratique peut mener à des cycles de développement allongés (un temps de mise sur le marché plus important) au cas où ces hypothèses ne sont pas vérifiées pour le RTOS cible considéré lors du déploiement.

Pour remédier à cette problématique, la première contribution de cette thèse a consisté en une prise en compte explicite des différentes hypothèses sur la plateforme logicielle considérées au niveau conception pour vérifier les contraintes de temps de l'application. Ceci a été traduit par la définition d'une plateforme logicielle abstraite d'analyse. Cette plateforme a la particularité d'être à la fois configurable et générique afin de s'adapter aux choix du concepteur d'une part (pour ne pas limiter ses choix) et assurer l'indépendance par rapport aux différents RTOS d'autre part.

Par la suite, en s'appuyant sur la première contribution, nous nous sommes intéressés à la définition d'un processus que nous avons appelé DRIM (Design Refinement toward Implementation Methodology).Ce processus se base essentiellement sur les modèles de plateformes logicielles c'est-à-dire le modèle de la plateforme abstraite utilisé pour des activités d'analyse au niveau conception et le modèle du RTOS cible sur lequel l'application sera déployée. Pour cela, les différents choix de modélisation ont été fixés et une mise en œuvre de ce processus a été proposée. Cette mise en œuvre a introduit les techniques nécessaires pour guider le déploiement des applications temps réel visées dans cette étude sur différents RTOS (ou sur plusieurs versions d'un même RTOS) en suivant la ligne du MDA d'une part et en assurant le respect des contraintes de temps de l'application à l'implémentation d'autre part.

Enfin, dans la dernière partie de notre thèse, nous avons procédé à une automatisation des différentes phases du processus au travers de la mise en place d'un outil qui a été intégré dans Qompass-Architect (outil CEA). Par la suite, nous avons testé l'applicabilité de notre

outil sur une application de contrôle d'un robot. Ceci nous a permis d'évaluer la capacité de l'outil proposé à détecter les descriptions de l'application non-implémentables pour un RTOS donné, ce qui présente l'avantage de réduire le temps de mise sur le marché d'une part et de guider l'utilisateur pour un choix approprié du RTOS cible d'autre part. Par ailleurs, cet outil est très bénéfique du fait qu'il permet de guider l'utilisateur pour trouver une description de l'application implémentable pour un RTOS donné en assurant toujours sa portabilité.

7.2 Perspectives

Les perspectives de ce travail de recherche sont multiples. Nous pouvons penser à considérer d'autres types d'analyse (analyse du pire temps d'exécution, analyse des performances, etc.) ce qui nécessitera la définition d'autres types de plateformes abstraites avec le niveau de détails approprié (granularité). Nous pouvons penser encore à considérer des architectures cibles sans des systèmes d'exploitation *OSless* architectures ce qui nous amènera à étudier/modéliser les techniques de parallélismes offerts par le matériel [40][41] afin d'évaluer la faisabilité de déploiement d'une application multitâches de haut niveau. Cependant, nous avons choisi de développer deux autres perspectives intéressantes de ce travail : La considération des architectures distribuées dans le but de rendre la méthodologie DRIM applicable pour des applications plus réalistes et l'exploitation des modèles de plateformes logicielles pour optimiser le déploiement d'une application temps réel selon différents critères.

Vers des architectures distribuées

Parmi les hypothèses que nous avons considérées sur les applications visées dans ce travail, est que l'architecture matérielle cible doit être de type monoprocesseur. Ceci limite en quelque sorte l'applicabilité de la méthodologie DRIM pour plusieurs types d'applications réelles basées sur des architectures multiprocesseurs voire distribuées telles que les applications du domaine automobile ou avionique [27][46]. Il serait ainsi intéressant d'étendre cette méthodologie pour supporter une large classe d'applications basées sur des architectures distribuées (ou multiprocesseur). Cela nécessite la définition d'autres types de test de faisabilité de déploiement, d'autres stratégies de portage et encore la proposition d'autres patrons. De plus, une telle extension requiert une modélisation plus détaillée des plateformes logicielles. Ainsi, d'autres ressources logicielles doivent s'ajouter aux modèles des plateformes (abstraite d'analyse et RTOS) telles que les ressources de communication.

Rétro-ingénierie des plateformes pour un déploiement optimisé des applications temps réel

Dans le contexte de développement des systèmes temps réel embarqué, la prise en compte des propriétés non-fonctionnelles est d'une importance majeure. En particulier,

l'étape de conception doit permettre la description l'application de façon à assurer le respect de ces propriétés à savoir les contraintes de qualité de service (QoS), la consommation mémoire, la consommation d'énergie, etc. Plusieurs travaux se sont intéressées à trouver les meilleurs choix architecturaux pour décrire une application dans le but d'optimiser une ou plusieurs de ces propriétés [58][19]. Cependant, les performances de ces choix peuvent varier d'une plateforme à une autre après le déploiement. Ainsi, une perspective de ce travail est d'ajouter une étape d'optimisation entre l'étape d'évaluation de faisabilité et l'étape de portage du processus DRIM. L'objectif de cette étape est de proposer à l'utilisateur la possibilité de changer les choix de la phase de conception dans le but d'optimiser une propriété non-fonctionnelle particulière pour le RTOS cible considéré. Pour cela, cette étape se base sur des informations décrites dans le modèle du RTOS (des poids sur l'utilisation des ressources par exemple). Par ailleurs, cette étape peut tirer profit des éléments configurables d'un RTOS de façon à l'adapter pour l'application dans le but d'avoir des meilleures performances en termes d'un critère particulier (comme le principe d'adéquation application/ architecture dans les approches co-design [56][71]).

Liste des Publications

1. **R. Mzid**, Ch. Mraidha, J-P. Babau, M. Abid. *RTOS-Aware Refactoring for Portable Real-Time Design Model*.Journal of Software (JSW).Accepté et à apparaître en juillet 2014.

2. **R. Mzid**, Ch. Mraidha, J-P. Babau, M. Abid. *Bridging the Gap between Real-Time Design Models and RTOS-Specific Models : A Model-Driven Approach*.Soumis à Software & Systems Modeling (Sosym), 2014.

3. **R. Mzid**, Ch. Mraidha, J-P. Babau, M. Abid. *A MDD Approach for RTOS Integration on Valid Real-Time Design Model.* The 38th Euromicro Conference On software Engineering and Advanced Applications (SEAA'12), Cesme, Izmir, Turkey, September 2012.

4. **R. Mzid**, Ch. Mraidha, J-P. Babau, M. Abid. *Real-Time Design Models to RTOS-Specific Models Refinement Verification*. The 5th International Workshop on Model Based Architecting and construction of Embedded Systems ACES-MB@MODELS 2012.Innsbruck, Austria, September 2012.

5. **R. Mzid**, Ch. Mraidha, A. Mehiaoui, S. Tucci Piergiovanni,J-P. Babau, M. Abid. *DPMP : A software Pattern for Real-Time Tasks Merge*. The 9th European Conference on Modeling Foundations and Applications (ECMFA'13). LNCS 7949, Springer, Montpellier, France, July 2013.

6. **R. Mzid**, Ch. Mraidha, J-P. Babau, M. Abid. *SRMP : A Software Pattern for Deadlocks Prevention in Real-Time Concurrency Models*.The 10th International ACM Sigsoft conference on Quality of software architectures (Qosa'14).Lille, France, July 2014.

Bibliographie

[1] *FreeRTOS*. http ://www.freertos.org/.

[2] *Wind River VxWorks RTOS*. http ://www.windriver. com/products/vxworks.

[3] *µITRON for small-scale embedded systems*. IEEE Micro, vol. 15, pp. 46–54, December 1995.

[4] *OSEK/VDX Time Triggered Operating System*. Specification version 1.0., July 2001.

[5] *Portable Operating System Interface (POSIX)*. The Open Group Base Specifications. ANSI/IEEE Std 1003.1, 2004.

[6] *Operating System Specification 2.2.3*. OSEK/VDX, February 2005.

[7] *Object Constraint Language (OCL)*. Object Management Group, OMG document number : formal/06-05-01, May 2006.

[8] *RTEMS Real Time Operating System*. http ://www.rtems.org, 2010.

[9] *NanoRK*. http ://www.nanork.org/projects/nanork/wiki/, 2011.

[10] *MicroC/OS-II*. http ://micrium.com/rtos/ucosii/overview/, 2012.

[11] *AUTOSAR*. http ://www.autosar.org/, 2013.

[12] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2) :159–192, 1982.

[13] J. P. A. Almeida. Model-driven design of distributed applications. In *On the Move to Meaningful Internet Systems 2004 : OTM 2004 Workshops*, pages 854–865. Springer, 2004.

[14] A. Antoniou. *Digital signal processing*. McGraw-Hill, 2006.

[15] T. Arpinen, M. Setala, E. Salminen, and T. D. Hannikainen. Modeling embedded software platform with a uml profile. In *Forum on Specification and Design Languages (FDL'07)*, September 2007.

[16] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36–41, 2003.

[17] C. Atkinson and T. Kühne. A generalized notion of platforms for model-driven development. In *Model-driven software development*, pages 119–136. Springer, 2005.

[18] W. Barnes. Arinc 653 and why is it important for a safety-critical rtos, 2004.

[19] C. Bartolini, G. Lipari, and M. Di Natale. From functional blocks to the synthesis of the architectural model in embedded real-time applications. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 458–467. IEEE, 2005.

[20] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.

[21] L. B. Becker, R. Holtz, and C. E. Pereira. On mapping rt-uml specifications to rt-java api : bridging the gap. In *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, pages 348–355. IEEE, 2002.

[22] A. BEN. Principles of concurrent programming. 1982.

[23] G. Bernat. Response time analysis of asynchronous real-time systems. *Real-Time Systems*, 25(2-3) :131–156, 2003.

[24] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2) :171–188, 2005.

[25] G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6) :47–54, 2000.

[26] A. W. Brown, D. J. Carney, E. J. Morris, D. B. Smith, et al. *Principles of CASE tool integration*. Oxford University Press, 1994.

[27] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006.

[28] M. Brun. Utilisation des techniques développées dans le cadre de l'ingénierie dirigée par les modèles pour la réalisation d'un outils de génération de code conforme osek/vdx à partir d'une description aadl. *rapport de Master*, 2006.

[29] M. Brun. *Contribution à la considération explicite des plates-formes d'exécution logicielles lors d'un processus de déploiement d'application*. PhD thesis, Nantes, 2010.

[30] W. E. H. Chehade. *Déploiement Multiplateforme d'Applications Multitâche par la Modélisation*. PhD thesis, Université Paris Sud-Paris XI, 2011.

[31] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next-generation design framework for platform-based design. In *Conference on using hardware design and verification languages (DVCon)*, volume 152, 2007.

[32] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate, and implement arinc653 systems using the aadl. In *ACM SIGAda Ada Letters*, volume 29, pages 31–44. ACM, 2009.

[33] J. Delatour, F. Thomas, G. Savaton, and S. Faucou. Modèle de plate-forme pour l'embarqué : première expérimentation sur les noyaux temps réel. *Actes des premières journées sur l'Ingénierie Dirigée par les Modèles (IDM 2005), Paris, France*, page 26, 2005.

[34] B. P. Douglass. *Real-Time Design Patterns : Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[35] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7) :1165–1178, 2008.

[36] J. Elloy. Editorial : Quelle informatique est donc nécessaire pour automatiser en temps réel. *Technique et Sciences Informatique*, 7(5) :395–396, 1988.

[37] J.-M. Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[38] C. Ferdinand, R. Heckmann, D. Kästner, K. Richter, N. Feiertag, and M. Jersak. Integration of code-level and system-level timing analysis for early architecture exploration and reliable timing verification. *Proceeding of the Embedded Real Time Software and Systems (ERTS2 2010). Toulouse, France*, 2010.

[39] J.-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In *Computer Aided Verification*, pages 85–96. Springer, 1993.

[40] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.

[41] J.-L. Gaudiot. The walls of computer design. In *Parallel and Distributed Processing and Applications*, pages 1–1. Springer, 2006.

[42] S. Gérard et al. Papyrus uml. *URL : http ://www. papyrusuml. org*, 2012.

[43] J. B. Goodenough and L. Sha. *The priority ceiling protocol : A method for minimizing the blocking of high priority Ada tasks*, volume 8. ACM, 1988.

[44] M. Guignard-Spielberg and K. Spielberg. Integer programming : State of the art and recent advances. *Annals of Operations Research 139*, 2005.

[45] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4) :42, 2008.

[46] B. Joseph. Les systèmes électroniques embarqués : un enjeu majeur pour l'automobile. In *Journées nationales de réflexion et de perspective sur les systèmes embarqués*, 2003.

[47] F. Jouault. Contribution à l'étude des langages de transformation de modèles. *Thèse de doctorat, Université de Nantes, France*, 2006.

[48] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, 1993.

[49] S. Kodase, S. Wang, and K. G. Shin. Transforming structural model to runtime model of embedded software with real-time constraints. In *Proceedings of the conference on Design, Automation and Test in Europe : Designers' Forum-Volume 2*, page 20170. IEEE Computer Society, 2003.

[50] H. Kopetz. The complexity challenge in embedded system design. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 3–12. IEEE, 2008.

[51] P. Kukkala, J. Riihimaki, M. Hannikainen, T. D. Hamalainen, and K. Kronlof. Uml 2.0 profile for embedded system design. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, pages 710–715. IEEE Computer Society, 2005.

[52] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209. IEEE, 1990.

[53] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM SIGMETRICS Performance Evaluation Review*, 14(1) :44–53, 1986.

[54] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[55] J. W. Liu. *Real-time systems*. Prentice Hall PTR, 2000.

[56] Y. Manai, J. Haggege, and M. Benrejeb. New approach for application architecture adequacy in hardware/software embedded system design. In *Faible Tension Faible Consommation (FTFC), 2011*, pages 39–42. IEEE, 2011.

[57] R. Marvie, L. Duchien, and M. Blay-Fornarino. Les plate-formes d'exécution et l'idm, chapitre 4. *Numéro ISBN*, pages 2–7462.

[58] A. Mehiaoui, E. Wozniak, S. Tucci-Piergiovanni, C. Mraidha, M. Di Natale, H. Zeng, J.-P. Babau, L. Lemarchand, and S. Gerard. A two-step optimization technique for functions placement, partitioning, and priority assignment in distributed systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 121–132. ACM, 2013.

[59] M. Mohamed, M. Romdhani, and K. Ghedira. Classification des approches de refactorisation des modèles. *IDM 2008*, page 169, 2008.

[60] A. Moore. Extending the rt profile to support the osek infrastructure. In *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, pages 341–347. IEEE, 2002.

[61] C. Mraidha. *Modelisation executable et analyse de proprietes temps reel*. PhD thesis, Evry-Val d'Essonne, 2005.

[62] C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard. Optimum : a marte-based methodology for schedulability analysis at early design stages. *ACM SIGSOFT Software Engineering Notes*, 36(1) :1–8, 2011.

[63] R. Mzid, C. Mraidha, A. Mehiaoui, S. Tucci-Piergiovanni, J.-P. Babau, and M. Abid. Dpmp : a software pattern for real-time tasks merge. In *Modelling Foundations and Applications*, pages 101–117. Springer, 2013.

[64] O.M.G. Uml profile for modeling and analysis of real time and embbeded systems (marte). *Object Management Group, http ://www.omg.org/marte/*, June.

[65] OMG. *MDA Guide Version 1.0.1*. OMG omg/2003-06-01, June 2003.

[66] O.M.G. Profile for schedulability, performance, and time specification (spt). *Object Management Group, http ://www.omg.org/SPT*, 2003.

[67] OMG. *MOF QVT final adopted specification*. http : //www.omg.org/docs/ptc/05-11-01.pdf, Novembre 2005.

[68] OMG. *Architecture analysis & design language (aadl), as5506*. Society of Automotive Engineer, 2006.

[69] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. January 2006.

[70] M. C. Paulk, C. V. Weber, B. Curtis, and M. E. CHRISSIS. *The capability maturity model : Guidelines for improving the software process*, volume 441. Addison-wesley Reading, 1995.

[71] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, 55(2) :99–112, 2006.

[72] A. Pinto. Metropolis design guidelines. rapport technique. *University of California, Berkeley*, page 30, 2004.

[73] M. Raphaël, D. Laurence, and B.-F. Mireille. Les plates-formes d'exècution et l'idm. In *Hermes, chapter 4*, pages 71–86.

[74] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A soc design methodology involving a uml 2.0 profile for systemc. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 704–709. IEEE, 2005.

[75] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *Design & Test of Computers, IEEE*, 18(6) :23–33, 2001.

[76] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-based development of embedded systems. In *Advances in Object-Oriented Information Systems*, pages 298–311. Springer, 2002.

[77] E. Seidewitz. What models mean. *Software, IEEE*, 20(5) :26–32, 2003.

[78] B. Selic. A generic framework for modeling resources with uml. *Computer*, 33(6) :64–69, 2000.

[79] B. Selic. On software platforms, their modeling with uml 2, and platform-independent design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 15–21. IEEE, 2005.

[80] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory : A historical perspective. *Real-time systems*, 28(2-3) :101–155, 2004.

[81] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9) :1175–1185, 1990.

[82] M. J. S. Smith. *Application-specific integrated circuits*, volume 7. Addison-Wesley Reading, 1997.

[83] J. Staunstrup and W. Wolf. *Hardware/software co-design : principles and practice*. Springer, 1997.

[84] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4) :110–111, 1997.

[85] A. S. Tanenbaum. *Structured computer organization*. Prentice Hall PTR, 1984.

[86] F. Thomas. *Contribution à la prise en compte des plates-formes logicielles d'exécution dans une ingénierie générative dirigée par les modèles*. PhD thesis, Université d'Evry-Val d'Essonne, 2008.

[87] Y. Trinquet and J.-P. Elloy. *Systèmes d'exploitation temps réel-Exemples d'exécutifs industriels*. Ed. Techniques Ingénieur, 2010.

[88] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE, 2007.

[89] G. Yang, M. Zhao, L. Wang, and Z. Wu. Model-based design and verification of automotive electronics compliant with osek/vdx. In *Embedded Software and Systems, 2005. Second International Conference on*, pages 7–pp. IEEE, 2005.

[90] R. Yemhalli. Real-time operating systems : An ongoing review. In *In Work-ln-Progress Sessions of the 21th IEEE Real-lime System Symposium (RTSSWIPOO)*, 2000.

[91] B. Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, Télécom ParisTech, 2008.

[92] H. Zeng, M. Di Natale, and Q. Zhu. Optimizing stack memory requirements for real-time embedded applications. In *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–8. IEEE, 2012.

[93] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9) :1250–1258, 2009.

[94] M. Zhao, Z. Wu, G. Yang, L. Wang, and W. Chen. Smartosek : a dependable platform for automobile electronics. In *The First International Conference on Embedded Software and System*, page 437, 2004.

Annexe A

Formulation MILP du patron DPMP

Cette annexe présente la formulation du patron de fusion des tâches de priorités distinctes (DPMP) en utilisation les techniques MILP (Mixed Integer Linear Programming)[44]. Par ailleurs, elle montre les résultats d'évaluation du programme linéaire proposé pour le DPMP et un extrait de la transformation qui permet de faire appel au solveur à partir du modèle.

Fonction objective

Un programme linéaire est basé sur une fonction objective qui permet de maximiser ou minimiser une expression qui formalise le problème.

$$\text{maximize} : \sum_{i,j \in \{1..m\}} \text{Merge}_{i,j} - \text{Utilization} \tag{A.1}$$

Dans cette expression, *Merge* est une variable booléenne qui permet de confirmer si deux tâches T_i et T_j sont fusionnées. En d'autres termes, si $Merge_{i,j}$ est égal à *1*, cela veut dire que la tâche T_i est fusionnée avec la tâche T_j. m représente le nombre des tâches dans le modèle de conception initial. Ainsi, cette expression permet de maximiser le nombre des fusions en minimisant l'utilisation du processeur (c'est-à-dire trouver la solution la plus optimisée pour fusionner les tâches en termes d'utilisation processeur).

Une fonction objective est généralement soumise à des contraintes qui permettent de limiter les solutions possibles (espace d'exploration). Autrement dit, les contraintes permettent d'éliminer les solutions non faisables pour le problème.

Les contraintes liées aux scénarios de fusion

Les contraintes présentées dans cette section permettent d'éliminer les scénarios de fusion non significatifs tels que la fusion d'une tâche avec elle-même ou la fusion d'une tâche déjà fusionnée.

$$n - \sum_{i,j \in \{1..m\}} \text{Merge}_{i,j} = N \tag{A.2}$$

$$\forall i, j \in \{1..m\}, \text{Merge}_{i,j} = 0 \quad \text{if} \quad (\text{isHarmonic}_{i,j} = 0) \quad \text{or} \quad (p_i = p_j) \tag{A.3}$$

$$\forall j \in \{1..m\}, \sum_{i \in \{1..m\} \wedge i \neq j} Merge_{i,j} \leq 1 \; ; \; \forall i,j,k \in \{1..m\} \wedge j,k \neq i, Merge_{i,j} + Merge_{k,i} \leq 1 \quad (A.4)$$

Dans l'expression A.2, n représente le nombre des niveaux de priorité distincts dans le modèle de conception initial et N représente ce même nombre autorisé par le RTOS pour l'application considérée. Ainsi, l'objectif de cette contrainte est de limiter le nombre de fusion jusqu'à N. Dans ce cas, le programme linéaire fusionne les tâches jusqu'à ce que *n devient égal à N*. La troisième contrainte A.3 a pour objectif d'interdire la fusion des tâches de priorités égales et les tâches non harmoniques. La contrainte A.4 permet d'empêcher la fusion d'une tâche déjà fusionnée.

$$\forall j \in \{1..m\}, TASKS_j = 1 - \sum_{i \in \{1..m\}} Merge_{i,j} \quad (A.5)$$

Dans expression A.5, la variable booléenne TASKS correspond au nouveau modèle de tâches après la fusion. En effet, si pour $i, j \in 1..m$ la variable $Merge_{i,j}$ est égale à 1, $TASKS_j$ sera égale à 0 et $TASKS_i$ sera égale à 1.

Les contraintes liées à l'aspect temps réel

Le modèle résultant après la fusion des tâches (modèle de conception modifié) doit respecter toujours les contraintes de temps de l'application qui sont exprimées dans les expressions A.6 et A.7 ci-dessous :

$$\forall i \in \{1..m\}, Rep_i \leq D_i \quad (A.6)$$

$$\text{utilization} \leq \text{Max\_Utilization} \quad (A.7)$$

Expression A.6 assure que le temps de réponse de chaque tâche dans le modèle résultant après la fusion est inférieur ou égale à son échéance. Par ailleurs, l'expression A.7 vérifie que l'utilisation processeur est inférieure à 1 comme condition nécessaire de faisabilité d'un modèle de tâches. La formule de calcul du temps de réponse est exprimée par la contrainte A.8 :

$$\forall i \in \{1..m\}, Rep_i = ffi_i + `_i + fi_i \quad (A.8)$$

Dans cette formule δ_i représente le temps d'exécution de la tâche T_i au pire cas. Ce terme est calculé comme suit :

$$\forall i \in \{1..m\}, \delta_i = TASKS_i * C_i + \sum_{j \in \{1..m\}} Merge_{i,j} * C_j \quad (A.9)$$

Dans cette expression, le temps d'exécution d'une tâche T_i supprimée suite à une fusion sera égal à 0 puisque $TASKS_i$ est égal à 0 d'une part et $\forall j \in 1..m, Merge_{i,j} = 0$ d'autre part. Par contre, le temps d'exécution d'une tâche qui absorbe d'autres tâches sera égal à la somme des temps d'exécution de toutes les tâches absorbées.

$$`_i = ı_i + fl_i \quad (A.10)$$

Le terme θ_i représente l'interférence de la tâche T_i avec les tâches les plus prioritaires que celle-ci. Nous désignons par HP_i l'ensemble des tâches plus prioritaires que T_i. Ce terme est calculé comme étant la somme de deux termes ζ_i et γ_i définis comme suit :

$$ı_i = TASKS_i * \sum_{\substack{j \in HP_i \\ j \in \{1..m\}}} TASKS_j * (\lceil \frac{Rep_i}{P_j} \rceil * C_j) \quad (A.11)$$

$$fl_i = TASKS_i * [\sum_{\substack{j \in HP_i \\ j \in \{1..m\}}} TASKS_j * (\sum_{k \in \{1..m\}} Merge_{j,k} * \lceil \frac{Rep_i}{P_j} \rceil * C_k)] \tag{A.12}$$

Le terme d'interférence est égal à 0 si la tâche correspondante est une tâche supprimée. Sinon, le calcul de ce terme prend en considération les différentes situations de fusion des tâches. En effet, du fait d'aspect non linéaire de ces deux expressions, le programme linéaire ne peut pas les interpréter. Ainsi, une linéarisation de l'expression A.11 est donnée par les expressions ci-dessous :

$$\forall i, j \in \{1..m\}, 0 \leq X_{i,j} - (\frac{Rep_i}{P_j}) < 1 \tag{A.13}$$

$$\forall i, j \in \{1..m\}, Y_{i,j} \leq X_{i,j} \; ; \; Y_{i,j} \leq M * TASKS_j \; ; \; X_{i,j} - M * (1 - TASKS_j) \leq Y_{i,j} \tag{A.14}$$

$$\forall i \in \{1..m\}, \imath_i \leq \sum_{\substack{j \in HP_i \\ j \in \{1..m\}}} Y_{i,j} * C_j \; ; \; \imath_i \leq M * TASKS_i \tag{A.15}$$

$$\forall i \in \{1..m\}, [\sum_{\substack{j \in HP_i \\ j \in \{1..m\}}} Y_{i,j} * C_j] - M * (1 - TASKS_i) \leq \imath_i \tag{A.16}$$

La linéarisation de l'expression A.11 nécessite en plus la définition de deux variables supplémentaires qui sont X et Y. En effet, la contrainte A.13 permet de calculer la valeur de $\lceil \frac{Rep_i}{P_j} \rceil$, par ailleurs, la contrainte A.14 calcule le terme $(TASKS_j) * \lceil \frac{Rep_i}{P_j} \rceil$. Les contraintes A.15 et A.16 permettent de déterminer la valeur finale de ζ_i. Le dernier terme dans la formule de calcul du temps de réponse est le temps de blocage β_i de la tâche résultante de la fusion exprimé comme suit :

$$\forall i \in \{1..m\}, fi_i = TASKS_i * BT_i \tag{A.17}$$

Ce terme est égal à 0 si la tâche correspondante est une tâche supprimée, sinon, ce le temps de blocage d'une tâche est donné par la contrainte suivante :

$$\forall i \in \{1..m\}, \quad BT_i = \begin{cases} B_i \quad \text{if} \quad \sum_{i,j \in \{1..m\}} Merge_{i,j} = 0 \\ \max_{j \in \{1..m\}} Merge_{i,j} * B_i \quad \text{Otherwise} \end{cases} \tag{A.18}$$

Ainsi, le temps de blocage d'une tâche sera égal au temps de blocage maximal de toutes les tâches fusionnées. Le temps de blocage d'une tâche qui n'est pas fusionnée sera égal à son ton de blocage initial (B_i défini comme un paramètre d'entrée). L'utilisation du processeur est calculée en se basant sur les trois contraintes ci-dessous :

$$\text{Utilization} \leq 1 \tag{A.19}$$

We define the Utilization term by the constraints just below :

$$\text{Utilization} = \bar{}_1 + \bar{}_2 \tag{A.20}$$

$$\bar{}_1 = \sum_{i \in \{1..m\}} TASKS_i * (\frac{C_i}{P_i}) \; ; \; \bar{}_2 = \sum_{i \in \{1..m\}} TASKS_i * \sum_{j \in \{1..m\}} Merge_{i,j} * (\frac{C_j}{P_i}) \tag{A.21}$$

Évaluation de la formulation MILP

Nous présentons dans cette section quelques évaluations du programme linaire qui décrit le patron DPMP. Les expériences sont réalisées sur un processeur Intel core i5-3360m

cadencé à 2,8 GHz avec 4 Go de la mémoire cache. CPLEX est utilisé comme un solveur MILP pour l'ensemble des expérimentations. Pour l'évaluation nous définissons en plus le terme cout comme suit :

$$\text{Cost} = \text{Current}_{\text{utilization}} - \text{Initial}_{\text{utilization}} \tag{A.22}$$

Ce terme référence la perte en termes de performance (ici en termes d'utilisation processeur) après l'application du patron. Cette courbe montre la variation du cout en fonction de l'application pour deux exemples de RTOS qui sont MicroC-OS/II (56 niveaux de priorité distinct pour une application) et Ecos configuré respectivement avec 16 et 8 niveaux de priorité distincts.

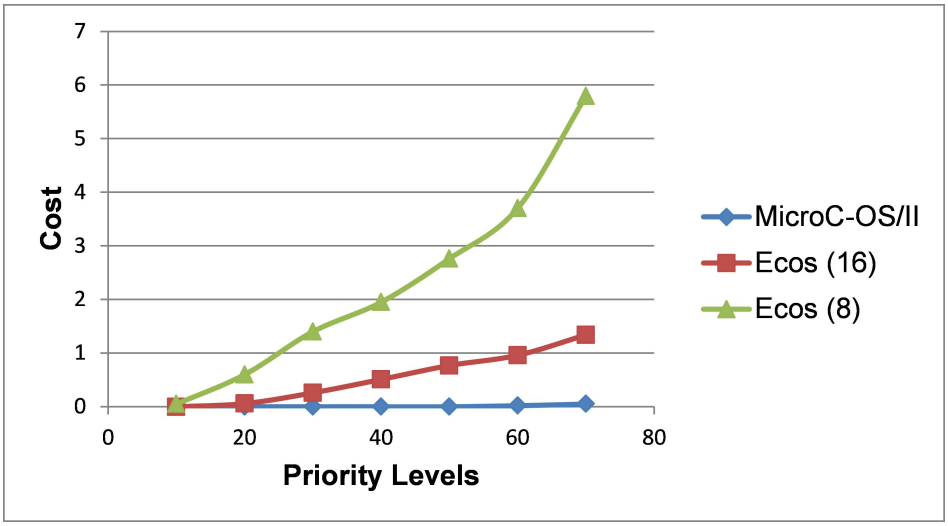


FIGURE A.1 – *Évaluation du patron DPMP*

Nous remarquons que cette variation (ou le coût) est plus important pour des applications (ou des modèles de conception) avec un grand nombre de niveaux de priorité distincts et pour des RTOS offrant un nombre plus limité des niveaux (le coût le plus élevé de perte de performance est obtenu pour Ecos configuré à 8 niveaux de priorité distincts seulement). La deuxième courbe A.2 montre la variation du temps d'exécution du programme linéaire en fonction de l'application. Cette courbe montre que ce le temps d'exécution est borné même pour des applications de grande échelle (de l'ordre de 70 niveaux de priorité distinct).

Extrait de la transformation d'appel du solveur

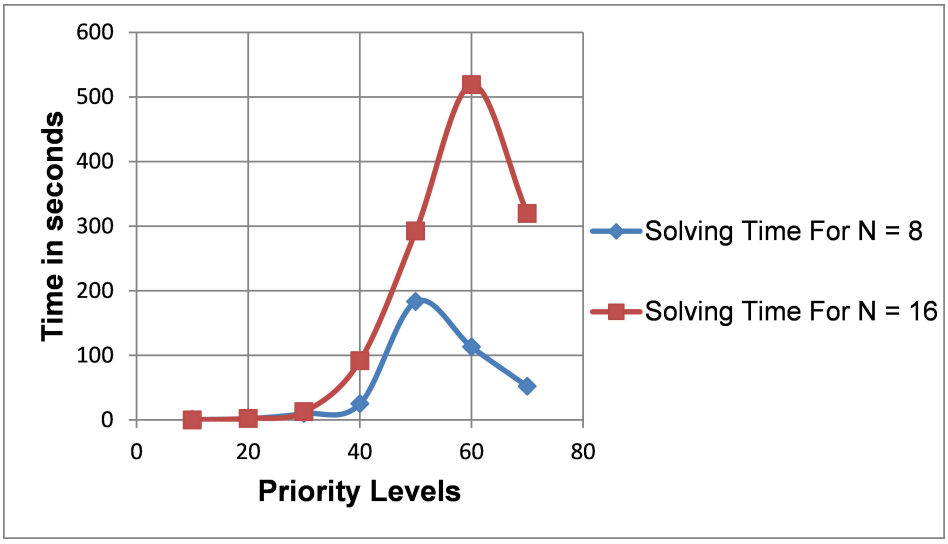


FIGURE A.2 – *Évaluation du temps de résolution du programme linéaire du patron DPMP*

```
//create a command to call the solver
String command [] = {"glpsol", "--model", "DPMP.mod ", "--data",
                          "DPMP.dat", "--output", "DPMP.txt" };
              Runtime r = Runtime.getRuntime();
              Process P = null;
              //execute the command
              try {
                     P = r.exec(command):
                     //Create a buffered reader from the Process
                     BufferedReader input = new BufferedReader(new
                     InputStreamReader(
                                   P.getInputStream()));
                 //Print the output into console
                     String line = null;
                     while ((line = input.readLine()) != null) {
                           System.out.println(line);
                     }
              } catch (IOException e) {
                     System.out.println("ERROR 1" + command [0] +
e.toString());
              }
```

FIGURE A.3 – *Extrait de la transformation d'appel du solveurpour l'exéution du programme linéaire du patron DPMP*

Annexe B

Modèles des plateformes logicielles

Cette annexe présente quelques exemples de modèles de plateformes logicielles élaborées pour évaluer la méthodologie DRIM proposée. Les différents modèles ont été créés en suivant les règles méthodologiques décrites dans le chapitre 5 de ce rapport. Par ailleurs, l'éditeur papyrus [42] est utilisé pour l'élaboration de ces différents modèles. Nous présentons tout d'abord le modèle de la plateforme d'analyse associée à l'outil Qompass-Architect(outil CEA). Ensuite, nous donnons les modèles des RTOS MicroC-OS/II [10], RTEMS [8], nano-RK [9] et AUTOSAR-OS [11].

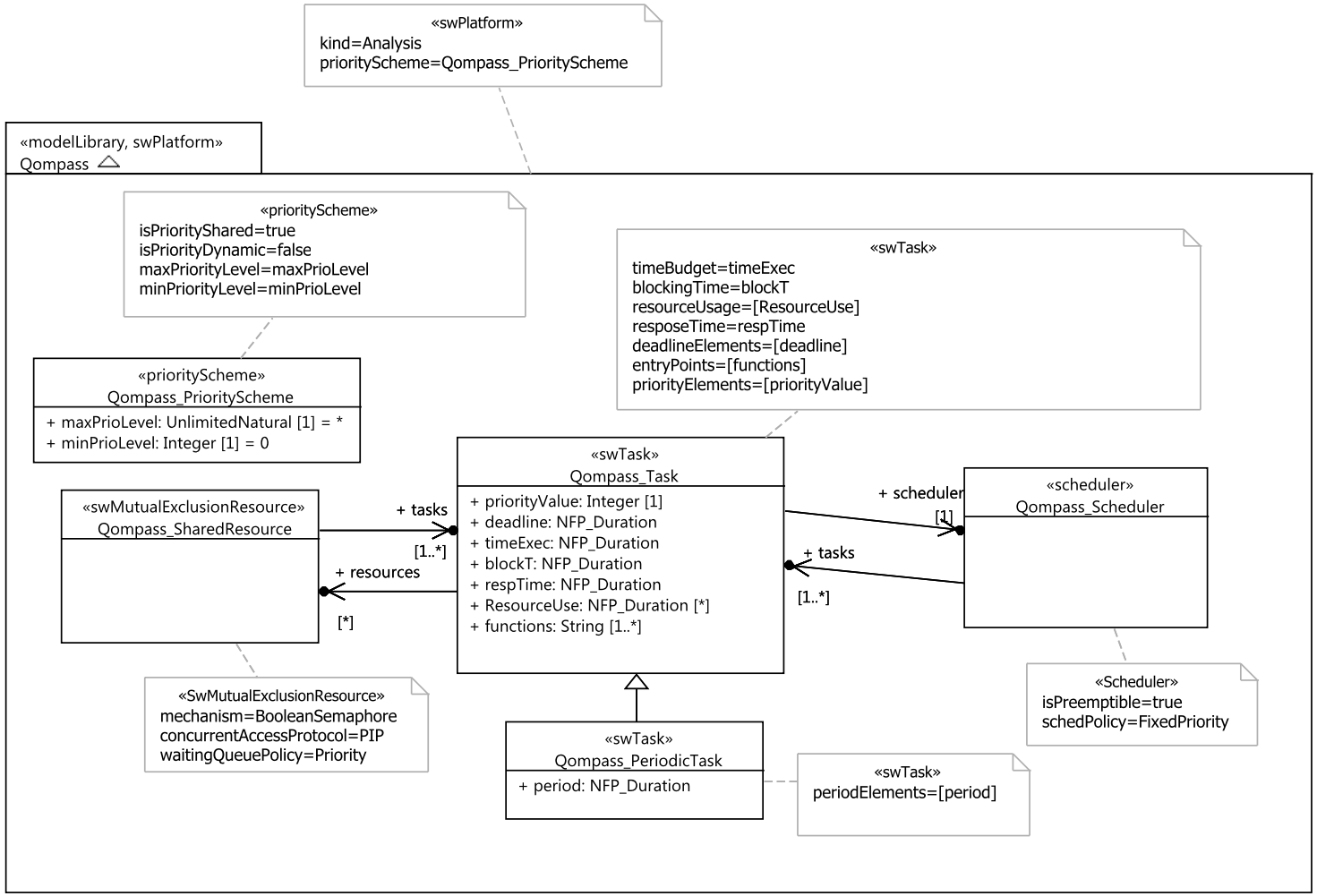


FIGURE B.1 – *Modèle d'une configuration possible de la plateforme abstraite d'analyse associée à Qompass*

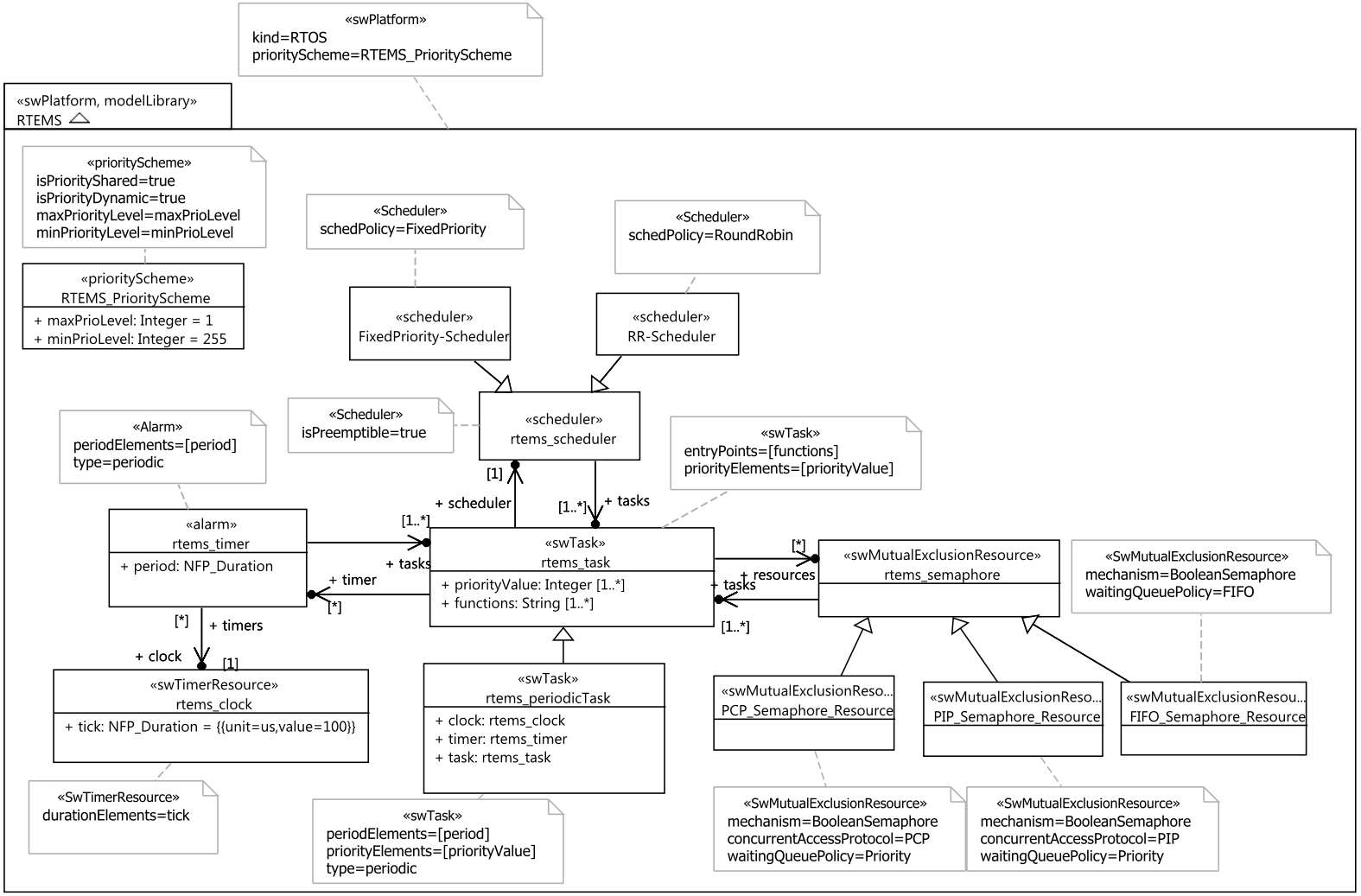


FIGURE B.2 – *Modèle de RTEMS*

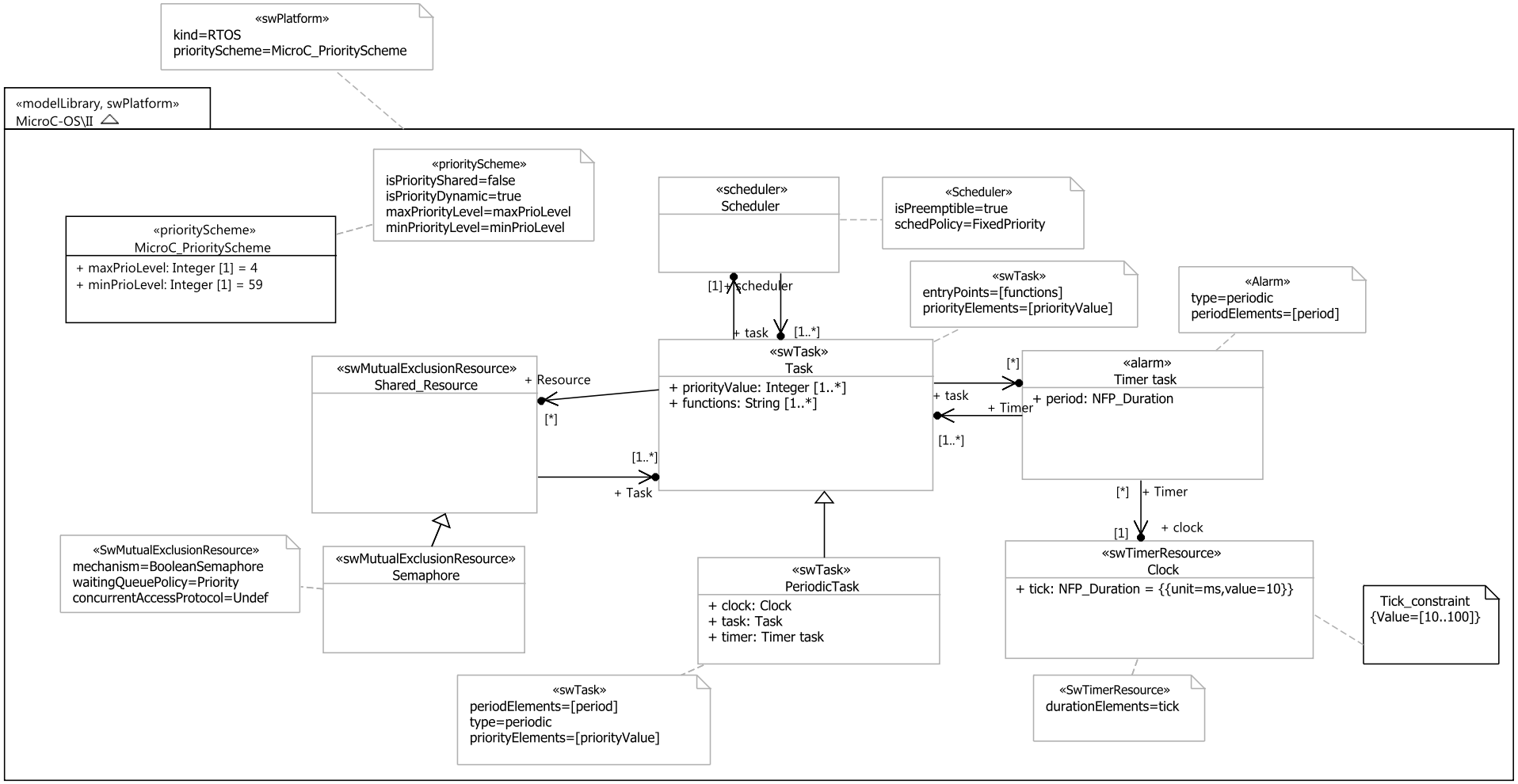


FIGURE B.3 – *Modèle de MicroC-OS/II*

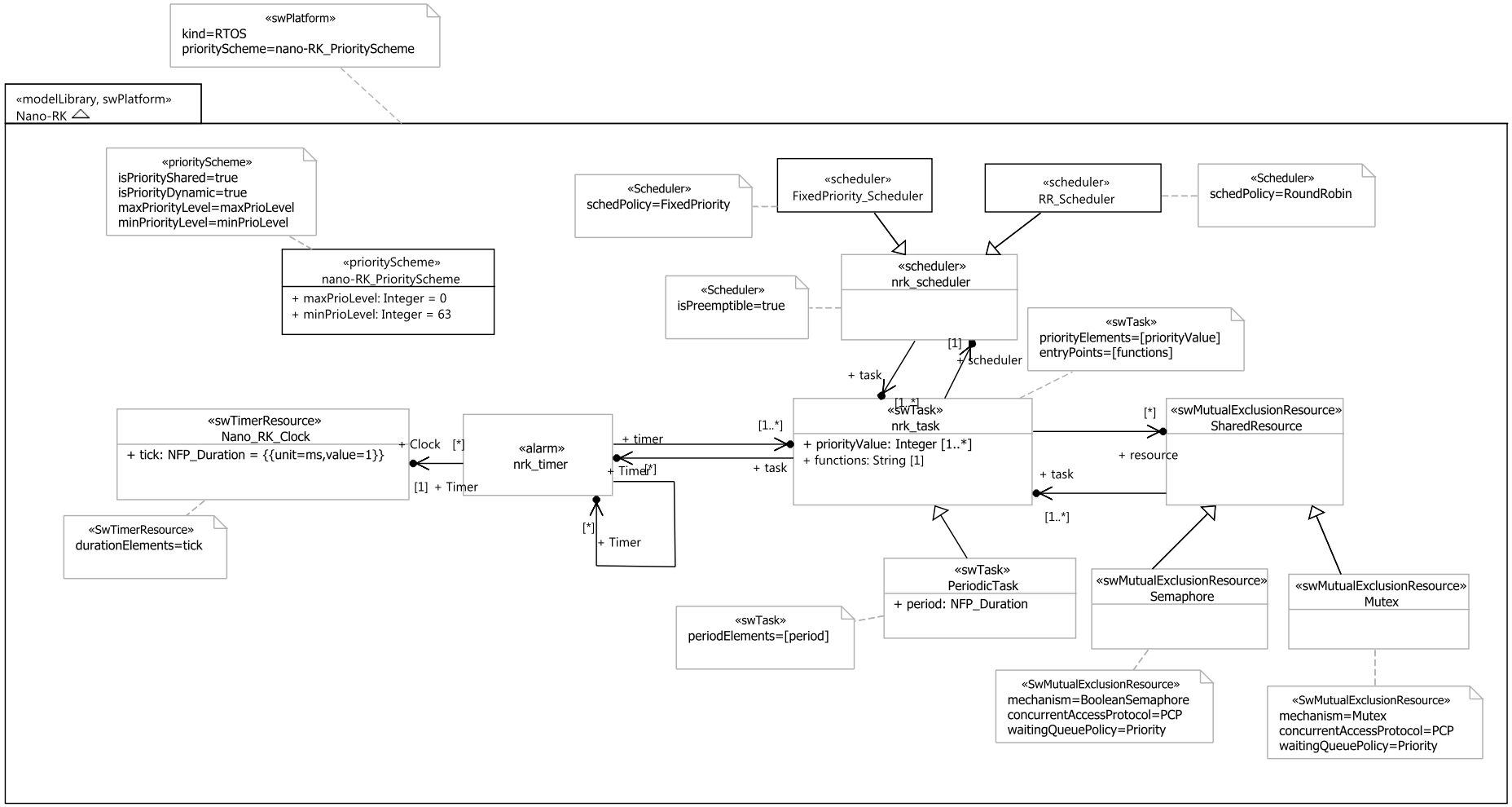


FIGURE B.4 – *Modèle de nano-RK*

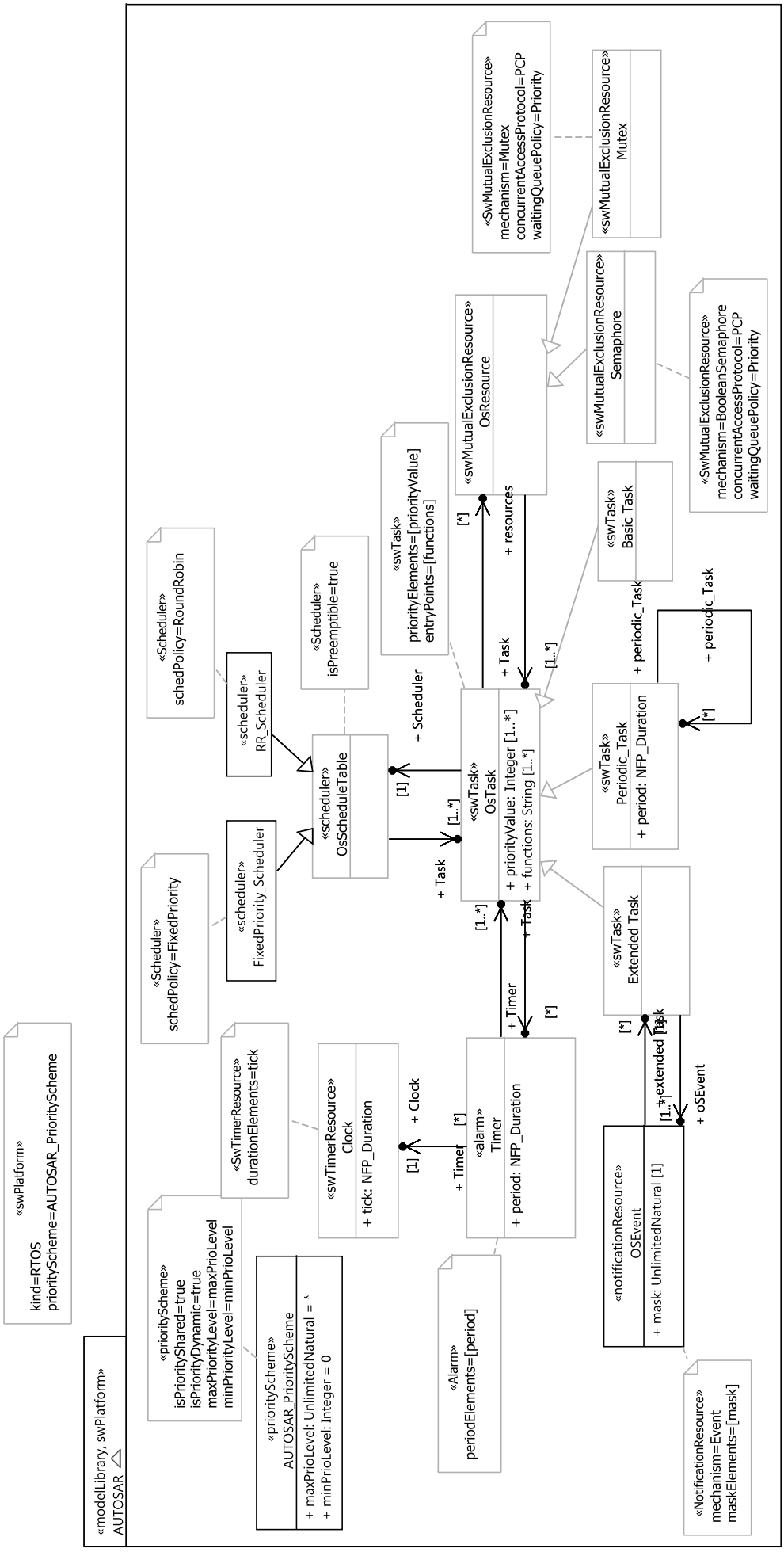


FIGURE B.5 – *Modèle de AUTOSAR-OS*